API REST y sistema de aprovisionamiento en containers para servIoTicy

Trabajo de final de grado

Sergi Alonso
sergi.alonso.garcia@est.fib.upc.edu

Director: David Carrera
Arquitectura de Computadores
dcarrera@ac.upc.edu

28 de Abril de 2017

Grado en Ingeniería Informática
Especialidad: Tecnologías de la Información

BARCELONA SUPERCOMPUTING CENTER (BSC)
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTECH

Resumen

Este documento recoge en detalle las contribuciones realizadas durante el trabajo de final de grado al proyecto servIoTicy, la plataforma de *stream processing* desarrollada por el departamento Data-centric Computing del Barcelona Supercomputing Center. Concretamente, se detalla el desarrollo de una API REST en Node.js y la integración de un sistema de aprovisionamiento usando Docker *containers* y Rancher, así como el conjunto de tareas que rodean al proyecto como el contexto, la planificación y el presupuesto.

Resum

Aquest document recull en detall les contribucions realitzades durant el treball de final de grau al projecte servIoTicy, la plataforma de *stream processing* desenvolupada per el departament Data-centric Computing del Barcelona Supercomputing Center. Concretament, es detalla el desenvolupament d'una API REST en Node.js i la integració d'un sistema d'aprovisionament utilitzant Docker *containers* i Rancher, aïxi com el conjunt de tasques que envolten el projecte com el context, la planificació i el pressupost.

Abstract

This document contains the details of the contributions made during the bachelor's thesis to the servIoTicy project, the stream processing platform developed by the Data-centric computing department of the Barcelona Supercomputing Center. Specifically, contains the detail of the development process of a REST API developed in Node.js and the integration of a provisioning system using Docker containers and Rancher, as well as the group of tasks that surround the project like the context, the planification and the budget.

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Introducción

1.1. ServIoTicy

En los últimos años se han ido perfeccionando las tecnologías que permiten la conexión ubicua sin cables de dispositivos inteligentes. Además, el análisis de datos en tiempo real ha permitido una nueva forma de aprovechar estos datos recolectados. Esto, combinado con el abaratamiento de la producción de sensores y sistemas embebidos, ha desembocado en una interconexión masiva de dispositivos físicos integrados en, por ejemplo, vehículos, electrodomésticos y muchas otras cosas más. Es lo que llamamos el Internet de las cosas, *the Internet of Things (IoT)*.

El número de dispositivos IoT conectados a Internet ha crecido significativamente. En 2015 se conectaron alrededor de unos 13.800 millones de dispositivos y los expertos estiman que que para 2020 puede llegar a los 38.500 millones [1].

Los dispositivos IoT generan flujos de datos desde muchas localizaciones que se reciben en una plataforma para ser tratados en tiempo real o almacenados para ser tratados más tarde. Pero al ritmo al que crecen estas interconexiones parece muy probable que cualquier entorno de gestión *IoT* tenga problemas de escalabilidad. Esto explica que haya una gran demanda de sistemas que ofrezcan una gestión de datos *IoT* que resulte escalable y suficientemente avanzada como para poder procesar un volumen de datos tan grande y una heterogeneidad de datos tan diversa.

Como solución a la convergencia entre los ecosistemas *Internet Of Things*, *Big Data* y *Stream Processing* surgió servIoTicy.

Este trabajo de final de grado se basa en una contribución a servIoTicy, un proyecto desarrollado por el equipo de Data-Centric Computing, del departamento Computer Sciences del BSC (Barcelona Supercomputer Center), con el que el autor colabora. ServIoTicy es una plataforma de almacenamiento y procesado de datos. Es una solución que integra capacidades de *multi-tenant data*[1] *stream processing*[2], una API REST (Application Programming Interface), analizado de datos y soporte

[1]Datos que pertenecen a diferentes propietarios.
[2]Procesado de flujos de datos en tiempo real.

multi protocolo en combinación con avanzadas tecnologías *data-centric*[2]. ServIoTicy pretende ser una platforma tecnológica que facilite la creación de servicios para que objetos, servicios y humanos puedan acceder a los datos creados por los dispositivos conectados a su plataforma.

1.2. Formulación del problema

Actualmente la plataforma servIoTicy está siendo remodelada con el objetivo de usar nuevas tecnologías que permitan una mejora en el rendimiento. De esta remodelación surgen una serie de tareas, algunas de las cuales se han seleccionado para desarrollar durante este trabajo. Estas tareas son:

- Renovación de la API REST: Muchas de las funcionalidades de servIoTicy han cambiado desde su remodelación y muchas de ellas afectan directamente a la API REST, ya sea por un cambio de especificación o por un cambio de versiones, por lo que hay que reprogramar muchas de sus funcionalidades. Partiendo de esto, el equipo decidió renovar completamente la API REST partiendo de cero usando otra tecnología que se adecue más a los requisitos de la nueva plataforma.
- Integrar un sistema de aprovisionamiento de la infraestructura en *containers*: Uno de los nuevos requisitos es que la plataforma sea desplegable en diferentes entornos, por ejemplo, en el entorno del cliente o como entorno de desarrollo. Esto daría una flexibilidad importante a la hora de realizar pruebas o incluso para un despliegue en producción. Además, debe ser un sistema basado en *Linux Containers*.
- Evaluar el rendimiento de la API REST: Una vez desarrollada la nueva API, es conveniente realizar un análisis de rendimiento del entorno para evaluar la solución.

1.3. Objetivos

De las tareas anteriores se extraen los siguientes objetivos para el trabajo de final de grado:

1. **Desarrollo de una nueva API REST para servIoTicy.**
2. **Integración de un sistema de aprovisionamiento en *containers* para servIoTicy.**
3. **Análisis de rendimiento de la API REST sobre la nueva plataforma.**

1.4. Introducción a las API REST

Una API (del inglés, Application Programming Interface) es un conjunto de procedimientos que realizan diversas funciones con el fin de ser utilizadas por otro software, es decir, es una interfaz que permite acceder a funcionalidades de otras aplicaciones sin tener que volver a desarrollarlas.

Muchos programas exponen sus API para permitir al usuario explotar las funcionalidades de forma programática. Sin embargo, en este trabajo de final de grado se habla de un tipo muy concreto de API, las API REST.

REST (del inglés, Representational State Transfer) es una estilo de arquitectura del *software* que define la interacción entre componentes dentro de un sistema *hypermedia* distribuido, como la *World Wide Web*. Una API REST, por tanto, provee de funciones que permiten al cliente hacer uso de un servicio web desarrollado por un tercero mediante el protocolo HTTP. Por ejemplo, la API REST de Twitter permite leer y escribir *tweets* o leer información de perfiles desde aplicaciones externas a Twitter.

Entre sus características se destaca el uso de una identificación uniforme de sus recursos con una URI, el uso de cabeceras HTTP tipificadas para cada tipo de operación y los mensajes autodescriptivos. Así, un usuario puede acceder a leer o escribir datos desde una aplicación externa de una forma intuitiva y fácil. Existen otros métodos que implementan servicios web como RPC, SOAP o WSDL, pero se intenta favorecer el uso de REST por ser más intuitivo y fácil de implementar. Durante este trabajo de final de grado se realizará una nueva API REST sobre Node.js para servIoTicy.

1.5. Introducción a los containers

Linux Containers[3] (LXC) es un método de virtualización para ejecutar múltiples sistemas Linux aislados sobre un mismo sistema operativo. Los *containers* desacoplan las aplicaciones de los sistemas operativos, empaquetando las piezas de software en un sistema de ficheros que contiene todo lo que necesita para ejecutarse. Docker[4] o LXD [5] son implementaciones de LXC. Durante este trabajo de final de grado se realizará la migración de servIoTicy a Docker *containers*.

1.6. Introducción a los sistemas de aprovisionamiento

Cuando se habla de aprovisionamiento en informática, se habla del proceso de desplegar y configurar los sistemas necesarios para dar un servicio. Esto se traduce en la automatización del despliegue de las herramientas y aplicaciones sobre el *hardware* que dará el servicio, sea cual sea, y esté donde esté.

Hace unos años esto se llevaba a cabo con tediosos *scripts* parametrizados que conectaban con las máquinas remotas y ejecutaban los procesos necesarios para adaptar la infraestructura. No obstante, con el tiempo, se han ido perfeccionando estos métodos para abstraer a los usuarios de las partes más complicadas. Ansible[6], Puppet[7] o Chef[8] son ejemplos de la evolución de estos métodos, en los que utilizando programación declarativa se crean plantillas para el aprovisionamiento de infraestructuras de forma fácil y intuitiva.

[3] `https://linuxcontainers.org/`
[4] `https://www.docker.com/`
[5] `https://linuxcontainers.org/lxd/`
[6] `https://ansible.com`
[7] `https://puppet.com/`
[8] `https://chef.io/`

No obstante, en este trabajo de final de grado se realizará un aprovisionamiento de de servIoTicy sobre Docker *containers*. Esto abre un abanico de herramientas muy interesantes, como por ejemplo *Docker-compose*, un lenguaje declarativo como los mencionados anteriormente, pero específico para desplegar containers sobre Docker. Durante este trabajo de final de grado se realizará un sistema de aprovisionamiento de servIoTicy basado en *Docker-compose*.

1.7. Estructura de este documento

El documento se divide en cinco partes:

La primera parte está dedicada al contexto del proyecto, su gestión y la planificación temporal y económica, así como un glosario de las tecnologías usadas en el proyecto.

La segunda parte está dedicada a los detalles relevantes del desarrollo de la API REST.

La tercera parte está dedicada a los detalles relevantes de la integración del sistema de aprovisionamiento en *containers*.

La cuarta parte está dedicada al análisis del rendimiento de la API REST.

La quina y última parte está dedicada a una conclusión global del proyecto.

Capítulo 2

Contexto del proyecto

2.1. Estado del arte

Aunque los distintos elementos a desarrollar en este Trabajo de Final de Grado forman parte de una unidad (el proyecto servIoTicy), para analizar el estado del arte de los elementos a estudiar es necesario dividirlo por partes:

2.1.1. API REST

En la última década, las empresas han empezado a exponer sus *APIs* para permitir a terceros construir funcionalidades nuevas. Las tecnologías tradicionales como *SOAP* (*Service-oriented architecture*) han ido evolucionando para reducir la interdependencia entre los elementos que la usan, terminando en la gran adopción de la arquitectura *REST* para diseñar servicios web. Gracias al crecimiento del uso de esta arquitectura, junto al de las tecnologías asociadas como *JSON* (*JavaScript Object Notation*), el desarrollo y uso de APIs ha acelerado de forma considerable, y servicios como Twitter, Netflix o Facebook procesan miles de millones de llamadas API al día [3].

Una de las razones por las que REST ha sido tan ampliamente aceptado es porque permite a los desarrolladores la habilidad de construir funcionalidades modulares con interfaces muy livianas que no requieren una compleja integración. Esto ha desembocado en la proliferación de *frameworks REST* para el desarrollo simple y rápido de APIs para la creación de servicios Web en distintos lenguajes de programación: Django REST framework[1], ExpressJS[2] y Slim[3] entre otros.

El estado actual del desarrollo de servicios web parece seguir apostando por las *API REST*, con la ayuda de frameworks que faciliten el trabajo.

[1] `http://django-rest-framework.org/`
[2] `http://expressjs.com/`
[3] `http://slimframework.com/`

2.1.2. Aprovisionamiento en *containers*

En la transición de la mayoría de servicios hacia el *cloud*, las operaciones de aprovisionamiento y despliegue de infraestructuras han ido evolucionando. La necesidad de escalar horizontalmente los servicios era la oportunidad perfecta para usar la flexibilidad que aportan los *containers*. Docker es la tecnología de virtualización de *containers* con más público y es probablemente el estándar *de facto* y esto ha ocasionado la proliferación de sistemas que orquestan este tipo de *containers*.

Es el caso de Kubernetes[4], un motor de orquestación de *containers* con una gran comunidad por detrás que ha ganado mucha fama en el último año [4] El proyecto ha sido desarrollado por Google con grandes contribuidores como Red Hat, CoreOS o Mesosphere. Amazon AWS ECS es también un gestor de containers, con una orientación más hacia el CaaS (Containers As A Service), gestionando las imágenes de Docker y orquestando su ejecución en sus propias instancias de computación. Algunas soluciones van más allá, DC/OS[5] se autodenomina sistema de operación distribuida. Está en una capa por debajo de la capa de orquestación mencionada antes, gestionando los recursos dinámicamente de los *clusters* que corren encima de él, siendo por tanto, complementaria a los *softwares* como Kubernetes para dar un paso más en el *scheduling* de recursos [5].

2.2. Actores

Por una parte, el proyecto va dirigido a los clientes del proyecto servIoTicy porque, a la larga, una vez puesto en producción, el proyecto beneficiaría a empresas TIC que implanten servIoTicy como plataforma de análisis de flujos de datos, reduciendo el tiempo de despliegue y facilitando su uso para que los clientes puedan concentrarse en el propio análisis de datos y no en la complejidad de la plataforma.

Por otra parte, el proyecto también va dirigido a los desarrolladores del equipo de servIoTicy, ya que este proyecto ofrece sistemas que facilitan algunas de sus tareas, como por ejemplo, la migración o nueva implantación de los servicios en diferentes entornos, así como la ejecución de pruebas o experimentos que actualmente tienen que hacer de forma manual.

2.3. Metodología y rigor

Este apartado pretende mostrar los elementos que forman el proceso de desarrollo, seguimiento y validación de este proyecto.

2.3.1. Metodología de trabajo

Este proyecto se lleva a cabo mediante una metodología *agile*, aunque sin ningún *framework* concreto como sería *SCRUM* o similares. La metodología de este proyecto se basa en la realización

[4] `https://kubernetes.io/`
[5] `https://dcos.io/`

de iteraciones sobre pequeñas divisiones del proyecto que minimizan la cantidad de planificación para las siguientes partes del proyecto. Cada división del proyecto, o cada *feature* no es añadida al proyecto hasta que no ha pasado por todas las fases, que son:

1. Análisis
2. Diseño
3. Desarrollo
4. Test

Al final de cada iteración se presenta la funcionalidad a los interesados, en este caso, al equipo de servIoTicy y en especial a su coordinador, que es el director de este proyecto, que validará la funcionalidad. Así se minimiza el riesgo y permite al producto a adaptarse a los cambios de forma rápida. Puede ser que más de una iteración sea necesaria para diferentes funcionalidades.

Para cada iteración o incluso para algunas fases de la misma, es necesaria una reunión con el equipo y/o el director. Normalmente éstas ocurren una vez a la semana si la agenda lo permite.

2.3.2. Metodología de seguimiento y validación

El seguimiento se lleva a cabo durante las reuniones del equipo, que se organizan mediante la aplicación *Slack*, una aplicación de mensajería que soporta *plug-ins* como calendarios y la capacidad de compartir código y gráficas de forma sencilla. En estas reuniones pueden realizarse demostraciones en tiempo real o discusiones más profundas sobre como afrontar algún problema en concreto.

A veces puede ser necesaria una evaluación del código. Si se diera el caso, el equipo cuenta con repositorios online de código fuente en *Github.com* dónde se carga la información del control de versiones *git*.

Por último, para cuantificar la calidad de la solución proporcionada por el autor, cabe recordar que uno de los objetivos de este trabajo de final de grado es la evaluación del rendimiento de la plataforma.

Capítulo 3

Planificación temporal

3.1. Duración estimada

La duración estimada del proyecto es de unos 6 meses, empezando el 17 de Octubre de 2016 y finalizando el 15 de Abril (incluyendo la memoria y documentación). Ésta planificación se divide principalmente en 4 partes:

- El desarrollo de la API REST
- La integración del sistema de aprovisionamiento.
- Un análisis de rendimiento.
- Documentar y escribir la memoria y la presentación.

3.2. Tareas

A continuación se detallan las tareas del trabajo de final de grado con tal de tener una visión más específica de cómo se han planificado.

3.2.1. Curso de gestión de proyectos (Curso GEP)

Curso de preparación para la gestión del proyecto. Se contextualiza y planifica el proyecto.

- Duración estimada: 64 horas.
- Asignación de rol: *Project manager*
- Recursos: Portátil proporcionado por el BSC.

3.2.2. API REST

1. Análisis de requisitos
    - Duración estimada: 24 horas.
    - Asignación de rol: Analista
    - Recursos: Portátil proporcionado por el BSC.
2. Diseño
    - Duración estimada: 40 horas.
    - Asignación de rol: Analista
    - Recursos: Portátil proporcionado por el BSC.
3. Implementación
    - Duración estimada: 184 horas.
    - Asignación de rol: Programador
    - Recursos: Portátil proporcionado por el BSC.
4. *Testing*
    - Duración estimada: 80 horas.
    - Asignación de rol: Programador
    - Recursos: Portátil proporcionado por el BSC.

3.2.3. Aprovisionamiento

1. Análisis de requisitos
    - Duración estimada: 40 horas.
    - Asignación de rol: Analista
    - Recursos: Portátil proporcionado por el BSC.
2. Diseño
    - Duración estimada: 16 horas.
    - Asignación de rol: Analista
    - Recursos: Portátil proporcionado por el BSC.
3. Integración
    - Duración estimada: 184 horas.
    - Asignación de rol: Programador
    - Recursos: Portátil proporcionado por el BSC.
4. *Testing*
    - Duración estimada: 56 horas.
    - Asignación de rol: Programador
    - Recursos: Portátil proporcionado por el BSC.

3.2.4. Análisis de rendimiento

1. Diseño del experimento
    - Duración estimada: 40 horas.
    - Asignación de rol: Analista
    - Recursos: Portátil proporcionado por el BSC.
2. Ejecución de los experimentos
    - Duración estimada: 56 horas.
    - Asignación de rol: Programador
    - Recursos: Portátil proporcionado por el BSC.
3. Tratamiendo de resultados
    - Duración estimada: 24 horas.
    - Asignación de rol: Programador
    - Recursos: Portátil proporcionado por el BSC.

3.2.5. Documentación

1. Memoria
    - Duración estimada: 120 horas.
    - Asignación de rol: *Project manager*
    - Recursos: Portátil proporcionado por el BSC.
2. Presentación
    - Duración estimada: 80 horas.
    - Asignación de rol: *Project manager*
    - Recursos: Portátil proporcionado por el BSC.

3.3. Diagrama de Gantt

La Figura 3.1 muestra la planificación de las tareas anteriores plasmada en un diagrama de Gantt.

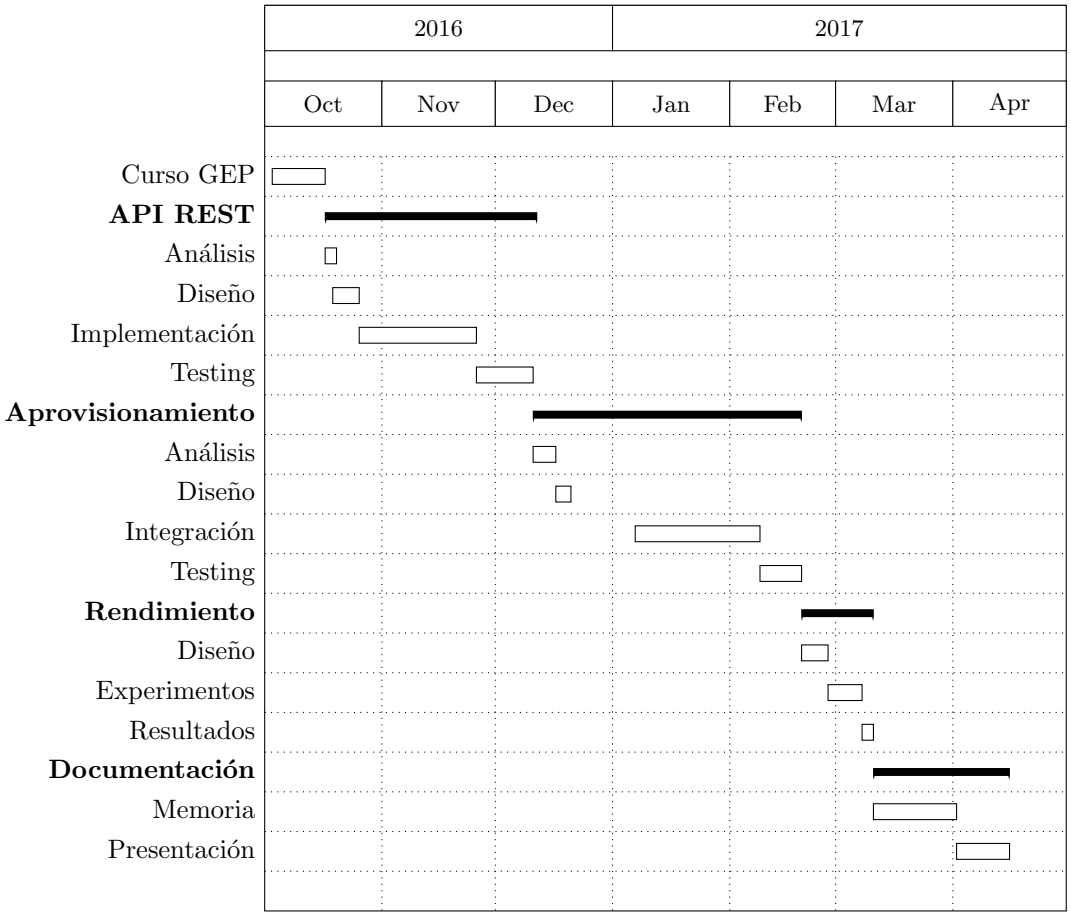


Figura 3.1: Diagrama de Gantt

3.3.1. Desviaciones

Cuando se realizó el curso de gestión de proyectos los objetivos no estaban acabados de definir, así que la planificación de las tareas era considerablemente diferente. Conforme el proyecto cogió forma, se acabó de definir la planificación que se ve en el diagrama 3.1. Esta planificación resultó ser bastante acertada (posiblemente porque se hizo de forma conservadora). No obstante, ha habido algunas desviaciones debido a agentes externos al proyecto como una baja por enfermedad o otros proyectos. Sin embargo, se ha conseguido cumplir con el *deadline* final reorganizando las horas disponibles según la planificación y al final no ha habido que eliminar ningún objetivo cumpliendo el *deadline* final usando el mismo número de horas.

Capítulo 4

Presupuesto

En el siguiente apartado se indica los distintos recursos que se utilizarán para el desarrollo del proyecto y sus costes asociados.

4.1. Coste recursos humanos

El valor económico del personal asociado al proyecto es:

Rol	Horas	Coste por hora (EUR)	Coste total (EUR)
Project manager	264	40	10560
Analista	160	30	4800
Programador	584	25	14600
Total	1008		29960

Tabla 4.1: Costes recursos humanos

4.2. Costes *hardware*

El valor económico de los recursos hardware teniendo en cuenta las horas que se han utilizado en el proyecto es:

Producto	Precio (EUR)	Unidades	Vida útil	Amortización (EUR)
Portátil Dell Latitude E7450	1500	1	4 Años	178,9
Total	1500			178,9

Tabla 4.2: Costes hardware

4.3. Costes *software*

El valor económico de los recursos software asociado al proyecto es:

Producto	Precio	Unidades	Vida útil	Amortización
Sistema operativo libre	0	1	1 Año	0
Sublime text	0	1	1 Año	0
SDK Couchbase	0	1	1 Año	0
Node.js	0	1	1 Año	0
Docker	0	1	1 Año	0
Rancher	0	1	1 Año	0
Total	0			0

Tabla 4.3: Costes software

4.4. Costes indirectos

Los costes indirectos asociados al proyecto son:

Producto	Precio (EUR/kWh)	Unidades (kWh)	Coste total (EUR)
Electricidad	0,10620	1008	107
Total			107

Tabla 4.4: Costes indirectos

4.5. Presupuesto total con impuestos

El valor total con impuestos asociado al proyecto es:

Tipo de coste	Precio (EUR)
Hardware	178,9
Software	0
Recursos humanos	29960
Indirectos	107
IVA 0.21	6351,63
Total	36597,53

Tabla 4.5: Impuestos

4.6. Desviaciones

Durante el curso de gestión de proyectos no se habían definido exactamente los objetivos, así que el presupuesto era considerablemente diferente. Al tener definido el proyecto se definió un nuevo presupuesto con una nueva cantidad de horas, que es el que se muestra en esta sección. No ha habido desviaciones en el presupuesto con respecto al expuesto en este documento.

Capítulo 5

Sostenibilidad

En este apartado se analiza la sostenibilidad en tres dimensiones distintas: económica, social y ambiental. Para cada una de ellas se analizan las cuestiones relevantes que pertenecen al hito inicial del proyecto tanto para su puesta en producción como para su vida útil. El informe se acompaña de una tabla donde se refleja la puntuación obtenida según la matriz de sostenibilidad.

5.1. Estudio de Impacto Económico

Sobre el impacto económico se debe destacar que se ha elaborado una evaluación de recursos y sus costes tanto personales como materiales (apartado anterior). No obstante, al ser un proyecto tan abierto no se ha tenido en cuenta el coste de reparaciones o actualizaciones una vez en producción ya que, cualquier funcionalidad agregada requeriría de su propio análisis temporal y de costes (cualquier reparación o actualización consistiría en gran parte en horas de recursos humanos, básicamente).

El autor cree que este proyecto parece una solución suficientemente económica si tuviera que ser competitiva frente a otros productos. De hecho, este proyecto está hecho en colaboración con el BSC y fomará parte de servIoTicy, el proyecto open-source candidato a ser un producto que salga a la venta, por tanto deberá ser competitivo si pretende tener éxito.

El equipo considera que no se podría haber estimado un mejor proyecto en menos tiempo con los recursos de los que se disponen. De hecho, el desarrollo se basa en sacar el máximo provecho a tecnologías existentes con tal de maximizar el esfuerzo de los desarroladores en tareas que requieran un alto grado de personalización hacia servIoTicy.

5.2. Estudio de Impacto Social

El Big Data ya forma parte de la sociedad, y cada vez más compañías se dedican al análisis de datos masivos. Herramientas como las que se pretenden crear en este proyecto pueden facilitar

la extracción de datos a diferentes empresas, sobretodo a pequeñas. Este proyecto surge de varias necesidades, la de poder usar los recursos de servIoTicy con una interfaz amistosa, y la de disponer de un sistema de despliegue que facilite su instalación y desarrollo en empresas de terceros, por lo tanto, este proyecto facilitará y mejorará la vida de sus consumidores. Ningún colectivo salga damnificado del desarrollo de este proyecto.

5.3. Estudio de Impacto Ambiental

Los recursos utilizados en este proyecto son mayoritariamente equipos que se donarán a organizaciones sin ánimo de lucro posteriormente. Para el proyecto no se necesita ningún tipo de materia primera, ni se fabrica ningún tipo de dispositivo, es puramente software.

Se estima que un ordenador puede generar unos 100gr de CO2 de media por hora, y un servidor el doble. Sin la existencia de este TFG probablemente se generarían más cantidades de CO2 haciendo uso de servIoTicy, ya que este proyecto trata de generar herramientas automatizadas eficientes para su uso. Sin estas herramientas, se seguirían usando técnicas más largas y laboriosas (más costosas) que aprovecharían menos los recursos de una máquina (o incluso varias), por tanto, generando más CO2.

Durante la explotación de este software se generará un impacto ambiental dependiendo de la repercusión y el uso que se dé de él, pero cabe destacar que este software ayuda a usar servIoTicy, que puede ser utilizado para análisis de datos medioambientales que pueden ayudar a realizar investigaciones relacionadas con la sostenibilidad y por tanto, disminuyendo potencialmente la contaminación.

Después del proyecto se seguirán usando los recursos que han formado parte del desarrollo, ya sea para otros proyectos o para donarlos a organizaciones, reduciendo así la huella ecológica.

5.4. Matriz de sostenibilidad

	PPP	Vida útil	Riesgos
Ambiental	7 [0:10]	14 [0:20]	0 [-20:0]
Económico	8 [0:10]	16 [0:20]	0 [-20:0]
Social	8 [0:10]	16 [0:20]	0 [-20:0]

Tabla 5.1: Matriz de sostenibilidad

Capítulo 6

Background de tecnologías

Antes de pasar a la implementación el autor considera apropiado describir el conjunto de tecnologías que se usan

6.1. Node.js

Node.js[1] es un entorno de ejecución libre que interpreta el lenguaje Javascript mediante el motor V8 de Google, utilizado también en el navegador Chromium[6]. Node.js tiene una arquitectura basada en eventos capaz de realizar entrada/salida de forma asíncrona. Así, gracias a su modelo de ejecución, una API escrita en Node.js puede aceptar una petición que requiere una operación de entrada/salida, prepararla para ser atendida por un flujo en segundo plano, y, sin esperar, atender a la siguiente petición (i.e. el flujo principal no se bloquea por entrada/salida). Cuando la primera ha sido completada, se emite un evento y es entonces cuando se recogen los resultados, sin necesidad de preguntar continuamente si la operación entrada/salida ya ha terminado. Esto permite tener una alta concurrencia de flujos que realicen entrada/salida sin bloquear las peticiones. Estas decisiones de diseño tomadas con el objetivo de mejorar el *throughput* y la escalabilidad se utilizan en aplicaciones web con muchas operaciones de entrada/salida así como en aplicaciones web en tiempo real. Netflix, Paypal, Uber, IBM y Microsoft son algunas de las grandes compañías que utilizan este entorno de ejecución para sus aplicaciones[7].

6.2. Express.js

Express.js[2] es un *framework* para aplicaciones web escrito en Javascript, publicado bajo licencia MIT diseñado para construir aplicaciones web y APIs. Es el *framework* estándard *de facto* para

[1] `https://nodejs.org/en/`
[2] `https://expressjs.com/`

Node.js. Su objetivo es ayudar a organizar las aplicaciones web en una arquítectura Modelo-Vista-Controlador, simplificando las tareas asociadas a la creación y gestión de rutas y peticiones [8].

6.3. MySQL

MySQL[3] es un sistema de gestión de base de datos relacionales (RDBMS), de licencia *open source* GNU GPL, y escrito en C y C++. MySQL es probablemente el motor de base de datos libre más conocido, y debida a su alta compatibilidad con la mayoría de sistemas (sobretodo Linux) se ha hecho muy popular a lo largo de los años. Es una opción ideal para gestionar una base de datos de un tamaño decente a un coste mínimo. Su fácil instalación permite tener una base de datos operativa en muy poco tiempo, y además resulta tener un muy buen rendimiento para el caso medio [9].

6.4. Couchbase

Couchbase [4], antiguamente Membase, es una base de datos NoSQL, *open-source* distribuida basada en documentos. Tiene la capacidad de distribuir la carga entre distintos nodos y proporcionar alta disponibilidad. Al contrario que las bases de datos relacionales, las bases de datos basadas en documentos guardan toda la información de un objeto en una única instancia en la base de datos, y cada objeto almacenado puede ser distinto de otro. Esto permite que el mapeo de objetos a una base de datos sea una tarea sencilla y lo hace atractivo para la programación de aplicaciones web.

La unidad básica de almacenamiento en Couchbase es un documento JSON asociado a una clave. El espacio se particiona en unidades lógicas llamadas *buckets*, que se distribuyen entre las máquinas del cluster mediante un mapeo compartido entre los servidores y las librerías del cliente. Así, couchbase proporciona un mecanismo de búsqueda basado en *key-value* (clave-valor) muy rápido y eficiente que le permite escalar de forma excelente [10].

Para trabajar sobre estas claves Couchbase proporciona distintos tipos de índices que pueden acelerar el acceso para según que tipos de consultas. Las más interesantes son:

1. Índice de vista de MapReduce, un indexador de vistas incrementales que usa una función *map reduce* escrita por el usuario (son pre-compiladas, lo que las hace útiles cuando hay que realizar consultas complejas que requieren muchos recursos).
2. Global Secondary Indexes: Son índices generados a partir de consultas que mezclan otros atributos, útil para indexar las consultas más complejas.

[3] `https://www.mysql.com/`
[4] `https://www.couchbase.com/`

6.5. Apache Kafka

Apache Kafka[5] es una plataforma de *stream processing* (procesado de flujos) desarrollada por Apache Software Foundation, desarrollada en Scala y Java. El objetivo de este *software* es proporcionar una plataforma unificada de alto *throughput*, baja latencia y gestión de datos en tiempo real, usando un modelo de publicación-suscripción. Es un recurso muy valorado para infraestructuras que procesan flujos de datos.

La arquitectura de Kafka está organizada alrededor de los siguientes términos: *topics, consumers, producers y brokers*. Todos los mensajes de Kafka están organizados en *topics*. Si se desea enviar un mensaje, se envía a un *topic* específico y si se desea leer, se lee desde un *topic* específico. Un *consumer* de *topics* pide los mensajes a Kafka, mientras que los *producers* envían mensajes a un *topic*. Por último, Kafka como sistema distribuído, corre en un cluster. Cada nodo se denomina un *broker*.

6.6. Apache Storm

Apache Storm[6] es un sistema de computación distribuido de datos en tiempo real. Está desarrollado en Clojure y fue adquirido y publicado por Twitter [11]. Storm es un cluster distribuido en el que hay dos tipos de nodos, *nimbus* y *supervisors*, *master y slave* respectivamente.

Storm ejecuta sistemas de computación en tiempo real sobre un *framework*. La lógica de estos sistemas se denomina *Topology* (topología) y debe enviarse a *nimbus* para que distribuya las tareas definidas en esa topología entre los diferentes *supervisors* para poder ser ejecutadas. Estas topologías tienen una estructura organizada básicamente en *spouts* (una fuente de flujos de datos) que leen datos de una fuente externa y las emiten hacia la topología, y *bolts* donde ocurre todo el procesado de datos y se ejecutan filtros, funciones, se comunica con bases de datos, etc [12].

6.7. Apache Zookeeper

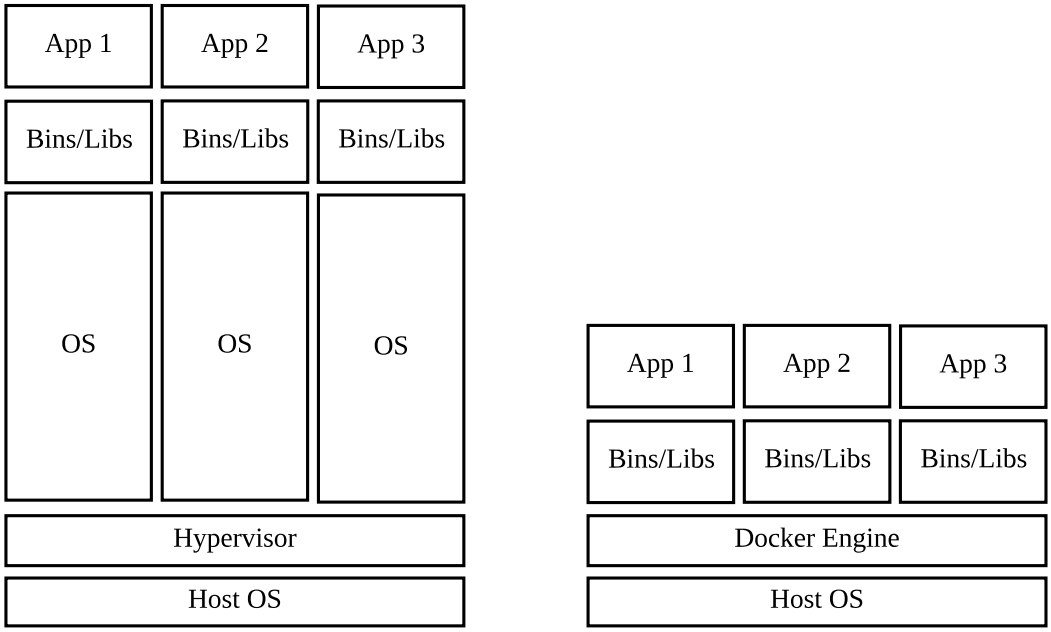
Apache Zookeeper[7] es un servicio centralizado para mantener la coordinación, configuración, proporcionar sincronización a equipos distribuídos y proporcionar servicios en grupo. Es útil para mantener todas estas características en sistemas distribuidos y no tener que realizar una implementación de estas carácterísticas cada vez. Zookeeper proporciona una única herramienta que las implementa todas.

[5] `https://kafka.apache.org/`
[6] `https://storm.apache.org/`
[7] `https://zookeeper.apache.org/`

6.8. Docker containers

Los *Linux Containers*[8] (LXC) es un método de virtualización para ejecutar múltiples sistemas Linux aislados sobre un mismo sistema operativo. Los *containers* desacoplan las aplicaciones de los sistemas operativos, empaquetando las piezas de software en un sistema de ficheros que contiene todo lo que necesita para ejecutarse. Esto se logra gracias a los mecanismos de aislamiento que proporciona el núcleo de Linux como *cgroups* y *namespaces* [13]. Docker[9] es una derivación de LXC desarrollada por Docker Inc, que proporciona aún más flexibilidad, convirtiéndose en la solución más popular para sistemas de virtualización sobre *containers*.

La gran ventaja de los *Docker containers* respecto a las máquinas virtuales es que proporcionan una capsula liviana y permite dejar atrás todo el *overhead* que supone trabajar con ellas, ya que con Docker solo se utiliza el sistema operativo del *host* donde se ejecuta, al contrario que con la virtualización clásica, en la que se ejecuta un sistema operativo por cada máquina virtual (ver figura 6.1).



(a) Virtualización con máquinas virtuales.

(b) Virtualización con containers.

Figura 6.1: Virtualización de máquinas virtuales vs. *containers*[14] (Adaptación).

Esto aporta flexibilidad para desplegar aplicaciones puesto que libera de la carga que supone ejecutar diversas instancias de un sistema operativo (que probablemente sean iguales) para cada aplicación, a la vez que mejora el rendimiento y aumenta la eficiencia.

Docker Inc desarrolla multitud de herramientas para trabajar con *containers*, como *Docker-*

[8] `https://linuxcontainers.org/`
[9] `https://www.docker.com/`

compose, un sistema de plantillas para definir conjuntos de servicios aprovisionados en *containers* para poder ser ejecutados de forma simple, incluso en un *cluster* si se usa un orquestador.

Además, también proporciona un repositorio centralizado de *containers* para que los usuarios y empresas publiquen las imágenes de sus software empaquetados. Este repositorio, llamado *Docker Hub*[10], cuenta con diversas funcionalidades incluyendo la construcción de *containers* en tiempo real en el *cloud* a partir del código de la aplicación.

6.9. Rancher

Rancher[11] es un orquestador (gestor de *cluster*) de Docker *containers*. Tiene la capacidad de aprovisionar servicios en *containers* a distintos *hosts* proporcionados por los propios usuarios y gestionar diferentes roles para los usuarios que vayan a desplegar esos servicios. Es compatible con *docker-compose* y diversos orquestadores.

Se organiza en *Stacks* (las plantillas de servicios) que se aprovisionan en los *hosts* añadidos a la infraestructura. Rancher mismo se ejecuta en un *container* y trae consigo un sistema de monitorización para los *hosts* en los que se ejecuta así como una interfaz gráfica servida como página web para poder controlar los servicios.

[10] `https://hub.docker.com/`
[11] `http://rancher.com/rancher/`

Capítulo 7

Renovación de la API REST de servIoTicy

En este apartado se detallan los aspectos más relevantes sobre el primer objetivo de este trabajo: la renovación de la API REST de servIoTicy.

7.1. Requisitos

A continuación se describen los detalles de la infraestructura y los requisitos que se extraen de ellos.

7.1.1. Funcionamiento e infraestructura de servIoTicy

El funcionamiento de servIoTicy se basa en la definición de tres categorías de datos:

1. *Streams* (flujos): Un usuario que desee publicar datos de cualquier tipo deberá definir previamente su estructura en un documento JSON. Ese documento JSON será almacenado y utilizado para comprender qué datos se están gestionando en cada momento en el procesador de datos.
2. *Subscriptions* (Suscripciones): Las *subscriptions* son documentos JSON donde se registra qué streams se quieren consumir, qué tipo de cálculo se quiere realizar (se pueden combinar con otros *streams*) y en qué *stream* aparecerá el resultado, pudiendo ser objeto de otras *subscriptions*.
3. *Updates* (Actualizaciones): Las *updates* son los datos publicados según la estructura definida en *streams*. Las *updates* tienen el mismo identificador que los *streams*.

Así, un dispositivo puede estar suscrito a cuatro *streams* diferentes, realizar un cálculo con ellos, y servir otro *stream* con los resultados. Por ejemplo, un usuario podría leer la temperatura de los distintos barrios de una ciudad (suscribiendose a cada uno de los sensores que publican la temperatura), hacer la media, y publicar un nuevo *stream* con la media de la ciudad, que a su vez podría ser utilizado para calcular la media de la comunidad autónoma, etc.

Una vez comprendido cómo están definidos los datos en la aplicación, es necesario entender cómo se gestionan esos datos y en qué tipos de aplicaciones se usan. Para ello, a continuación se describen los componentes y funcionalidades relevantes para el desarrollo de la API (la numeración corresponde a la Figura 7.1).

1. Los dispositivos conectados a la infraestructura de servIoTicy envían y reciben *updates* al sistema. Para ello, necesitan una interfaz que permita gestionar las *subscriptions* y las definiciones de *streams*, así como un sistema de identificación única que permita identificar tanto los dispositivos como los flujos de datos. Esta interfaz es la API REST, que recibirá peticiones HTTP siguiendo un formato concreto que se explica más adelante.

2. La API REST, una interfaz de comunicación entre los dispositivos y las diversas tecnologías de servIoTicy. Su renovación es uno de los objetivos de este trabajo y debe contar con las carácterísticas necesarias para poder ingerir y enviar la mayor cantidad de datos posible a los dispositivos conectados. Debe, también, proteger esta transmisión sin perder demasiado rendimiento en el proceso. Para ello debe controlar la comunicación con los distintos elementos de la infraestructura y gestionar de forma correcta los datos que pasan a través de ella. La tecnología de esta API en la versión anterior de servIoTicy era un *backend* desarrollado en Java, y para la nueva versión el equipo del proyecto ha pedido que se desarrolle sobre Node.js.

3. Una base de datos de autenticación, donde se guarda únicamente todo lo relativo a la identificación de los dispositivos del sistema y aislada de la otra base de datos. En la versión anterior de servIoTicy, esta base de datos es y mantenerla entra dentro de los requisitos.

4. Un sistema de colas que gestiona la publicación de datos de los dispositivos y mantiene el estado en caso de fallo de las bases de datos o el sistema de procesado. Esta tecnología viene predefinida, y es irremplazable. La API deberá tratar con ella sí o sí: Kafka.

5. Un sistema de computación en tiempo real que realizará las computaciones necesarias de las *updates* según las definiciones de los usuarios y luego almacenará los resultados en la base de datos. Este sistema es un software diseñado por el departamento que corre sobre Apache Storm. Esta tecnología también viene predefinida y es irremplazable pero no afecta directamente al desarrollo de la API porque la conexión se realiza a través de Apache Kafka.

6. Una base de datos para almacenar los datos destinados a la computación del sistema y que está aislada de la base de datos que contiene los datos de los usuarios, relativos a la autenticación. Esta tecnología también viene predefinida y es irremplazable: Couchbase.

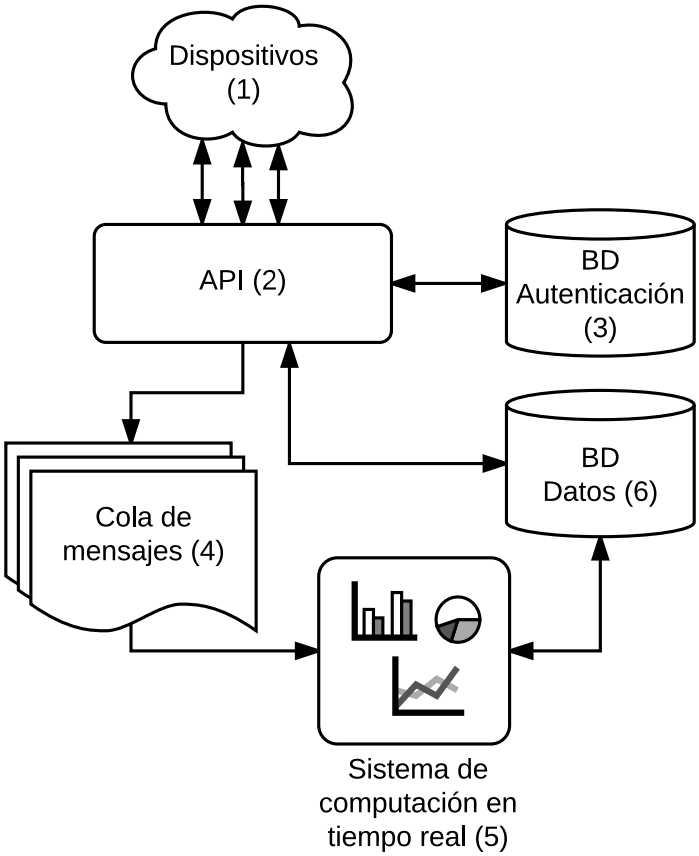


Figura 7.1: Diagrama de la infraestructura.

Nota: Por simplicidad, se ha elidido el software Zookeeper debido a que es un elemento que sirve para mantener un control de estado de Kafka y Storm y no afecta en ningún aspecto al desarrollo de la API REST (es completamente transparente para el desarrollador). En el apartado dedicado al sistema de aprovisionamiento sí que se tiene en cuenta dada su relevancia en la infraestructura.

7.1.2. Node.js y Express.js

Como se ha visto en los apartados anteriores, servIoTicy se basa en el envío y recepción de muchos datos que acaban siendo leídos o escritos en bases de datos. Eso significa mucha entrada/salida, algo que puede suponer un problema si no se busca una solución preparada para lidiar con estos entornos. Por ello, el equipo require que el *backend* de la API sea Node.js, una tecnología que sigue el modelo *event-driven* y *non-blocking I/O*, es decir, un sistema basado en eventos que no se bloquee por entrada/salida[15]. Este modelo es altamente eficiente y muy escalable porque el servidor está aceptando peticiones continuamente sin esperar a que las operaciones de entrada/salida terminen, y encaja perfectamente con las necesidades de la API de servIoTicy, que estará constantemente realizando operaciones entrada/salida.

Adicionalmente, para facilitar el diseño y el mantenimiento del código se precisa el uso del framework Express.js.

7.1.3. Enrutamiento REST

Para la API se pide mapear funciones a direcciones REST. El modelo de URL REST (Transferencia de Estado Representacional) es un estilo de arquitectura de software para sistemas hipermedia distribuidos como la World Wide Web. Esto significa que los métodos y rutas usadas deben ser autodescriptivas para cada recurso (e.g `GET api/streams/` devolverá una lista de *streams*, `POST api/streams/0894` insertará el stream 0894 a la lista de streams), que se usen cabeceras específicas HTTP para cada método (ver Tabla 7.1 [16]) y que los mensajes que se envíen como respuesta sean autodescriptivos. Cada método se utiliza dependiendo de la operación a realizar.

HTTP Verb	**CRUD**
POST	Crear
GET	Leer
PUT	Actualizar/Reemplazar
PATCH	Actualizar/Modificar
DELETE	Eliminar

Tabla 7.1: Referencia de métodos REST

7.1.4. Soporte de *micro-batching*

Micro-batching es una optimización que se require para mejorar el rendimiento de servIoTicy. Consistiría en dar soporte para que el cliente pudiera agrupar los mensajes antes de enviarlos a la API con el objetivo de reducir la cantidad de cabeceras necesarias para enviar un mismo conjunto de datos. El micro-batching es un mecanismo necesario y utilizado en la mayoría de *software* dedicado al stream processing. Es el punto medio entre agrupar una gran cantidad de datos o no agrupar ninguna, es decir, cuenta con la ventaja de tener una latencia menor que un batching elevado pero un *throughput* inferior. Soportando micro-batching se da la habilidad al cliente de enviar un conjunto de *updates* en vez de una solo, aumentando el throughput a costa de la latencia. Es por tanto, cuestión de encontrar la relación adecuada entre estas dos variables. Esta relación se buscará en el capítulo dedicado a las pruebas de rendimiento, el último objetivo.

7.1.5. Resumen de requisitos

Con todo lo descrito en los anteriores apartados, se pueden definir los requisitos que deberá cumplir la solución que el autor propone en el apartado de diseño:

- Desarrollo en Node.js: La API requiere Node.js para poder contar con las ventajas del paradigma *event-driven* y *non-blocking I/O*.

- Uso del framework Express.js: Para facilitar el diseño y el mantenimiento del código en el futuro, se precisa el uso del *framework* Express.js.
- Uso de rutas estilo REST: Las URL a las que se hacen las peticiones deben seguir lo máximo posible el modelo REST para conseguir direcciones a recursos de forma autodescriptiva.
- Gestión de los datos de servIoTicy: La API debe proporcionar la capacidad de gestionar los diferentes tipos de datos de servIoTicy: *Streams*, *subscriptions* y *updates*.
- Gestión de usuarios: La API debe proporcionar las funciones necesarias para la gestión de los usuarios de servIoTicy.
- Sistema de autenticación: La solución propuesta debe permitir la autenticación y protección de los datos de diferentes propietarios.
- Conexión cifrada: La conexión entre las diferentes partes debe ser cifrada para proteger la privacidad de los usuarios de servIoTicy.
- Micro-batching: La API es, por su naturaleza, probablemente el cuello de botella de la aplicación. Cuanto mejor sea su rendimiento más se aprovechará el resto de la infraestructura. Se requiere usar micro-batching como optimización de la comunicación entre la API y el cliente.
- Integración con MySQL, Kafka y Couchbase: Estas tres tecnologías vienen predefinidas y habrá que lidiar con ellas. Es esencial buscar una solución que permita una integracion con ambas tecnologías de la forma menos problemática posible siempre que se cumpla el resto de los requisitos.

Comprender los requisitos de la infraestructura de servIoTicy es esencial para poder diseñar una pieza de *software* con la que complementar el proyecto. Ahora ya es posible realizar un diseño de la solución.

7.2. Diseño

A continuación se detallan las decisiones de diseño tomadas durante la fase del desarrollo de la API.

7.2.1. Arquitectura de módulos

Como se ha visto en la especificación, los datos de ServIoTicy están divididos en tres tipos: *Streams*, *Subscriptions* y *Updates*. Por esta razón, para la API se propone un diseño por módulos separandolos dependiendo de los datos que se gestionen, es decir, uno para cada tipo de datos. Además, habrá otros módulos adicionales necesarios para la autenticación y la gestión de usuarios, que se verán más adelante. El resultado es una arquitectura muy simple que permitirá un desarrollo rápido y con pocos problemas (ver Figura 7.2).

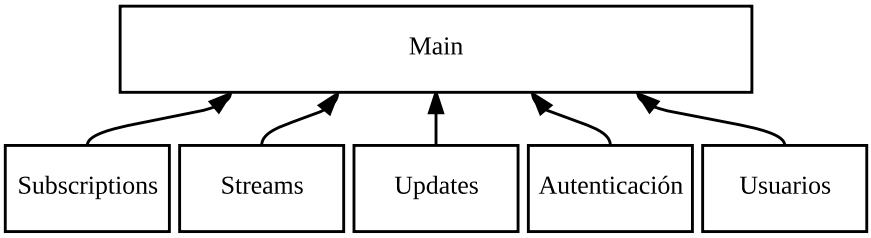


Figura 7.2: Arquitectura de módulos.

7.2.2. Rutas REST para la gestión de *Streams*

Para la gestión de *Streams* se proponen las rutas siguientes:

Nota Todas las rutas usan el identificador del usuario que está realizando la acción. Para mantener el modelo REST, es esencial que este identificador sea implicito en la petición y no forme parte de la URL. Esto se puede conseguir pasándolo como cabecera HTTP o, en este caso, mediante el *token* que se utilizará para la autenticación.

- Crear *streams*: Un método que permite la creación de *streams* en el sistema. La definición del *stream* en sí debe ser el cuerpo de la petición HTTP. La respuesta debe ser el identificador del *stream* creado. El método HTTP designado es POST.
- Consultar *streams*: Un método que permite la consulta de todos los *streams* de un mismo propietario. No es necesario ningún identificador ni un cuerpo en la petición. La respuesta debe ser el conjunto de *streams* creados por el usuario de identificador implícito. El método HTTP designado es GET.
- Consultar *streams* compartidos: La definición de los *streams* en servIoTicy permite escoger una *whitelist* y una *blacklist*. Con este método se permite el listado de los *streams* a los que se tiene acceso para poder consumir sus datos. La respuesta debe ser el listado de *streams* que pueden ser consumidos por ese tenant. El método HTTP designado es GET.
- Consultar *stream*: Un método que permite la consulta de un *stream* cuyo identificador formará parte de la URL. La respuesta debe ser el *stream* correspondiente al identificador. El método HTTP designado es GET.
- Modificar *stream*: Un método que permite la modificación de un *stream* cuyo identificador formará parte de la URL. Para modificar un *stream* se incluirá la nueva definición en el cuerpo de la petición. La respuesta debe ser un mensaje de confirmación. El método HTTP designado es PUT.
- Eliminar *stream*: Un método que permite la eliminación de un *stream* cuyo identificador formará parte de la URL. La respuesta debe ser un mensaje de confirmación. El método HTTP designado es DELETE.

Acción	Ruta	Método
Crear streams	`/api/streams`	POST
Consultar streams	`/api/streams/`	GET
Consultar compartidos	`/api/streams/shared`	GET
Consultar stream	`/api/streams/{ID}`	GET
Modificar stream	`/api/streams/{ID}`	PUT
Eliminar stream	`/api/streams/{ID}`	DELETE

Tabla 7.2: Resumen de rutas y métodos para funcionalidades de gestión de *streams*.

7.2.3. Rutas REST para la gestión de *Subscriptions*

Para la gestión de *Subscriptions* se proponen las rutas siguientes:

Nota Todas las rutas usan el identificador del usuario que está realizando la acción. Para mantener el modelo REST, es esencial que este identificador sea implicito en la petición y no forme parte de la URL. Esto se puede conseguir pasándolo como cabecera HTTP o, en este caso, mediante el *token* que se utilizará para la autenticación.

- Crear *subscriptions*: Un método que permite la creación de *subscriptions* en el sistema. La definición de la *subscriptions* en sí debe ser el cuerpo de la petición HTTP. Para la creación de una *subscription* es necesario que el *stream* origen y *stream* destino existan y sean válidos. La respuesta debe ser el identificador de la nueva *subscription*. El método HTTP designado es POST.
- Consultar *subscriptions*: Un método que permite la consulta de todas los *subscriptions* de un mismo propietario. No es necesario ningún identificador ni un cuerpo en la petición. La respuesta debe ser el conjunto de *subscriptions* creadas por el usuario de identificador implícito. El método HTTP designado es GET.
- Consultar *subscription*: Un método que permite la consulta de una *subscription* cuyo identificador formará parte de la URL. La respuesta debe ser la *subscription* correspondiente al identificador. El método HTTP designado es GET.
- Modificar *subscription*: Un método que permite la modificación de una *subscription* cuyo identificador formará parte de la URL. Para modificar una *subscription* se incluirá la nueva definición en el cuerpo de la petición. La respuesta debe ser un mensaje de confirmación. El método HTTP designado es PUT.
- Eliminar *subscription*: Un método que permite la eliminación de una *subscription* cuyo identificador formará parte de la URL. La respuesta debe ser un mensaje de confirmación. El método HTTP designado es DELETE.

Acción	**Ruta**	**Método**
Crear *subscriptions*	`/api/subscriptions`	POST
Consultar *subscriptions*	`/api/subscriptions/`	GET
Consultar *subscription*	`/api/subscriptions/{ID}`	GET
Modificar *subscription*	`/api/subscriptions/{ID}`	PUT
Eliminar *subscription*	`/api/subscriptions/{ID}`	DELETE

Tabla 7.3: Resumen de rutas y métodos para funcionalidades de gestión de *susbcriptions*.

7.2.4. Rutas REST para la gestión de *Updates*

Para la gestión de *Updates* se proponen las rutas siguientes:

- Consultar *update*: Un método que permite la consulta del último dato introducido al *stream* con la misma identificación. Es necesario el identificador del *stream*, que formará parte de la URL del recurso. La respuesta debe ser el contenido de la última *update*. El método HTTP designado es GET.
- Insertar *update*: Un método que permite la inserción de *updates* al *stream* del mismo identificador. Se deben insertar los datos como cuerpo de la petición, ya sea en conjuntos de micro-batching o en objetos simples (ver Figura 7.3). Con micro-batching damos la capacidad al cliente de realizar una única petición HTTP para el envío de un conjunto de *updates*, disminuyendo el número de peticiones necesarias para enviar el mismo número de datos a costa de la latencia.

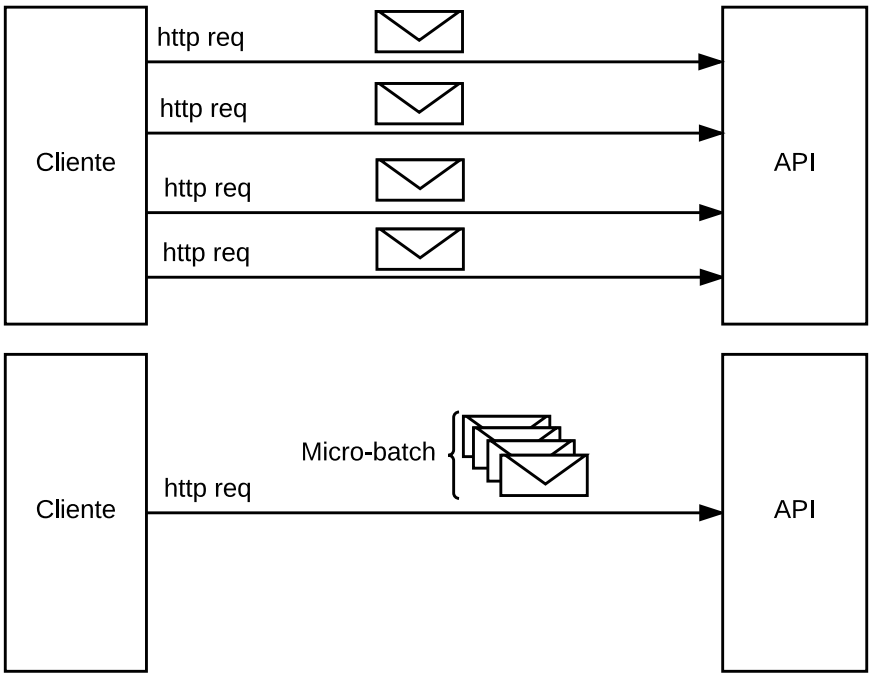


Figura 7.3: Comunicación estándar y con micro-batching.

Acción	**Ruta**	**Método**
Consultar *updates*	`/api/updates/{ID}`	GET
Insertar *updates*	`/api/updates/{ID}`	PUT

Tabla 7.4: Lista de rutas y métodos para funcionalidades de gestión de *updates*.

7.2.5. Rutas REST para la gestión de Usuarios

Uno de los requisitos para la API es que se puedan gestionar sus usuarios como administrador. Para ello, se han desarrollado las funciones necesarias para poder crear, consultar y eliminar usuarios que tienen acceso a la API. No obstante, no entra dentro de este trabajo la diferenciación entre los usuarios normales y administradores porque aún se están tomando decisiones acerca de los roles que va a haber dentro del proyecto.

- Crear usuario: Un método que permite la creación de usuarios en el sistema. La definición del usuario en sí debe ser el cuerpo de la petición HTTP. La respuesta debe ser un mensaje de confirmación. El método HTTP designado es POST.
- Consultar usuario: Un método que permite la consulta del *username* de un usuario dado su identificador, que formará parte de la URL. La respuesta debe ser el *username* correspondiente al identificador del usuario. El método HTTP designado es GET.

- Eliminar usuario: Un método que permite la eliminación de un usuario cuyo identificador formará parte de la URL. La respuesta debe ser un mensaje de confirmación. El método HTTP designado es DELETE.

Acción	**Ruta**	**Método**
Crear usuario	`/api/users`	POST
Consultar usuario	`/api/users/{ID}`	GET
Eliminar usuario	`/api/users/{ID}`	DELETE

Tabla 7.5: Lista de rutas y métodos para funcionalidades de gestión de usuarios.

7.2.6. Rutas REST para la autenticación de usuarios.

La API requiere un sistema de autenticación. Una de las formas más sencillas y aceptadas para llevar a cabo esta función es con el sistema de intercambio de *JSON Web tokens*, que se basa en la transferencia de información entre dos partes de forma compacta y encriptada, siendo un objeto JSON la información a encriptar[17].

Este mecanismo consiste en entregar al usuario un *token* con el que hacer las llamadas a la API. Este *token* tiene una validez y caducidad determinada, periodo tras el que deberá renovarse el *token*, es decir, pedir otro a la API. Es una forma sencilla y elegante de diseñar una autorización y una forma fácil de transferir información sensible entre las partes en el momento de la autenticación de los usuarios.

- Autenticación: Un método que permita al usuario identificarse mediante usuario y contraseña. Estos campos deben ir en el cuerpo de la petición HTTP, y la respuesta será el *token* que deberá usarse en las subsecuentes peticiones. El método HTTP designado para esta operacion es POST.

Acción	**Ruta**	**Método**
Autenticación usuario	`/api/auth/`	POST

Tabla 7.6: Lista de rutas y métodos para funcionalidades de autenticación de usuarios.

7.2.7. Usuarios e identificadores únicos

El diseño de los usuarios es muy sencillo: una estructura de datos que pueda almacenar un identificador único, un nombre de usuario y una contraseña. El identificador debe ser un UUID (Identificador único universal) y la contraseña debe estar encriptada mediante una función de *hash*.[1] como SHA256 o SHA512.

Los usuarios deberían contar con los siguientes atributos:

[1] Una función que puede usarse para mapear datos de tamaño arbitrario a datos de tamaño fijo.

- Identificador (UUID)
- Nombre de usuario
- *hash* de la contraseña

7.2.8. Conexión cifrada sobre TLS

Para la conexión cifrada se usará HTTP sobre TLS (Transport Layer Security)[18]. TLS usa lo que se denomina criptrografía asimétrica, o criptografía de llave pública. Este es un sistema estándar de cifrado en el que el servidor encripta con la llave privada la información enviada, y el cliente debe usar su correspondiente clave pública para desencriptarlo, corroborando así su identidad. Al ser un proyecto en desarrollo, bastará con usar un certificado auto-firmado, para poder construir la infraestructura necesaria para soportar comunicación encriptada. Una vez el proyecto esté en producción, sería recomendable registrarlo en una Autoridad de Certificación.

7.3. Implementación

A continuación se describe la implementación y el funcionamiento de distintas areas del proyecto:

7.3.1. Puntos relevantes sobre el desarrollo en Node.js

La API se ha desarrollado en Javascript con Node.js y Express.js. Esto ha complicado en algún punto el desarrollo de la aplicación. Esto se de debe a que al no ser bloqueante, en el código no se puede depender de que una aplicación funcione dependiendo de un valor de retorno de otra función, puesto que es posible que el valor esté o no asignado en el momento en que se ejecute (ver Listado 7.1), dependerá de si la operación se ha completado o no.

```
1 function imprimirValor() {
2     //Pedimos un resultado a la base de datos (operacion I/O).
3     var res = queryMySQL(SELECT *);
4     //Imprimir por pantalla la variable res.
5     //En este punto no se asegura que 'res'.
6     //tenga el valor esperado.
7     console.log(res);
8 }
```

Listado 7.1: Ejemplo de código no bloqueante en Node.js

Para solucionar esto, en los módulos de Node.js se acostumbran a utilizar los *callbacks* de Javascript, que son funciones que se ejecutan en el momento en que se señaliza que la operación ha

terminado gracias al sistema basado en eventos. Los *callbacks* consisten en pasar como parámetro la función que se ejecutará cuando la operación termine (ver Listado 7.2).

```
1 queryMySQL(SELECT *, function(res) {
2       //Esta función solo se ejecutará cuando
3       //la query a MySQL haya terminado.
4       //En este punto sí se asegura que 'res'
5       //tenga el valor esperado.console.log(res););
```

Listado 7.2: Ejemplo de callback en Node.js

Como se puede observar, la estructura del código cambia radicalmente. Escribir código así dista mucho de las prácticas tradicionales y no es intuitivo, y ha requerido un tiempo de adaptación importante dentro de la implementación de la API.

7.3.2. Puntos relevantes sobre el desarrollo sobre Express.js

Express.js es un *framework* diseñado para hacer enrutamiento de peticiones HTTP. Se define enrutamiento a la gestión de como una aplicación responde a una petición dependiendo de la URL a la que se dirige. Para ello, de instanciarse un *Router* y a continuación se definen los métodos que se ejecutarán cuando se llegue a una ruta en concreto (ver ejemplo en Listado 7.3).

```
1  // Incluir el módulo Express e instanciar el router.
2  var express = require('express');
3  var router = express.Router();
4  // definir la ruta con el método GET: birds.com/
5  router.get('/', function (req, res) {
6    res.send('Birds home page')
7  });
8  // definir la ruta con el método POST: birds.com/about
9  router.post('/about', function (req, res) {
10   res.send('About birds')
11 });
```

Listado 7.3: Ejemplo de enrutamiendo en Express.js

En el desarrollo de esta API, se crea un *Router* por cada módulo de rutas, uno para cada tipo de datos (ver sección 7.3.4).

7.3.3. Puntos relevantes sobre la integración con Couchbase

Para realizar la conexión se ha utilizado el módulo de Node.js *couchbase*. Este módulo implementa una interfaz para Node.js con la que poder desarrollar (es lo que llaman un Software Development Kit). El módulo es oficial e incluye soporte para conectar con los diferentes *bucket* donde se almacenan los distintos tipos de datos de servIoTicy.

Este módulo implementa las denominadas N1QlQuery, unas consultas escritas en N1QL, un lenguaje declarativo que extiende SQL a JSON [19], con las que se realizan las distintas consultas a la base de datos de Couchbase. Aunque también se pueden hacer consultas a Couchbase simplemente proporcionando una clave (Couchbase usa un modelo *key-value*), es necesario usar N1QlQuery si se quieren utilizar los *Global Secondary Indexes*, una indexación a un nivel superior de la indexación normal de claves de Couchbase, necesaria para ofrecer un mejor rendimiento a las consultas más complejas de más de un atributo.

7.3.4. Arquitectura de módulos

En Node.js existe la posibilidad de empaquetar las librerías en módulos. Es una forma sencilla de dividir funcionalidades y definir una estructura sencilla y reaprovechable de las librerías. Tal y como se diseñó la API, se pueden mapear fácilmente los tipos de datos que utiliza servIoTicy a distintos módulos en Javascript. Esto en Express.js se traduce en la siguiente estructura de dependencias (ver Figura):

- routes/auth.js: Módulo que implementa las rutas de autenticación.
- routes/streams.js: Módulo que implementa las rutas de gestión de *streams*.
- routes/subscriptions.js: Módulo que implementa las rutas de gestión de *subscriptions*.
- routes/updates.js: Módulo que implementa las rutas de gestión de *updates*.
- routes/users.js: Módulo que implementa las rutas de gestión de *usuarios*.
- middleware/auth.js: Módulo que implementa el sistema
- app.js: Módulo principal de la aplicación (Main).
- cbutils.js: Módulo de utilidades de couchbase que se utilizan en diversos módulos.
- utils.js: Módulo de utilidades de diversas que se utilizan en diversos módulos.

7.3.5. Rutas REST para la gestión de *Streams*

A continuación se describen los métodos de gestión de *streams*. La definición de los streams en Couchbase son de formato libre y es independiente de la implementación de la API exceptuando los campos: *tenant*, *whitelist* y *blacklist*.

- Crear *streams*: La implementación de este método es simple, basta con realizar una inserción mediante el SDK de Couchbase al *bucket* de *streams* introduciendo un nuevo UUID generada por el servidor y la definición del *stream* como cuerpo de la petición. Esto generará un documento en couchbase con el siguiente formato (ver Listado 7.4):

```
1  {
2    "channels": {
3      "count": {
4        "type": "number"
5      },
6      "offset": {
7        "type": "number"
8      }
9    },
10   "tenant": "fdd234-919f-4b682f6d--a1be477b-f66035",
11   "blacklist":["a1be477b-2f6d-4b68-919f-f66035fdd234"]
12 }
```

Listado 7.4: Definición de un *stream* en servIoTicy

- Consultar *streams*: Para este método hay que realizar una N1qlQuery a Couchbase. Concretamente una consulta que devuelva el documento (`meta(streams).id`) en el que el campo *tenant* (propietario) sea el mismo que el identificador implícito en el token.
- Consultar *streams* compartidos: Para la compartición de *streams* se han indexado los campos *whitelist* y *blacklist* de todas las definiciones de los *streams*. Gracias a esta indexación, para tener los resultados deseados es suficiente con realizar la siguiente N1qlQuery:
  - `WHERE ID in whitelist OR ID not in blacklist`,

  que retornará el identificador del documento (`meta(streams).id`) en esos campos.
- Consultar *stream*: La consulta de *streams* se realiza mediante una N1QlQuery idéntica a la de consultar *streams* salvo por filtrar el resultado con el identificador del *stream*.
- Modificar *stream*: La modificación de *streams* se realiza mediante la inserción de una distinta definición dentro del mismo documento, cuya identificación se recibe por URL. La implementación es idéntica a la de crear *streams* salvo por el uso del mismo identificador, sin generar uno nuevo.
- Eliminar *stream*: La eliminación de los *streams* es un método más complicado debido a que no pueden quedar *subscriptions* a un *stream* inexistente, por tanto se deben buscar todas las referencias al *stream* y eliminar esas subscripciones. La dificultad de este método radica en el asincronismo de Node.js. No se puede borrar el *stream* hasta que se hayan borrado tosas sus suscripciones, pero no se sabe el momento exacto en que va a ocurrir eso. Para solucionar este problema, se utiliza la librería `async`, que lanzará un evento cuando se borren todas las suscripciones que se hayan encontrado y ejecutará una *callback* que borrará ese *stream*.

7.3.6. Rutas REST para la gestión de *Subscriptions*

A continuación se describen los métodos de gestión de *subscriptions*. La definición de las *subscriptions* en Couchbase son de formato libre y es independiente de la implementación de la API exceptuando los campos: *origin* y *destination*, que deberán ser los identificadores de los *streams* de origen y destino.

- Crear *subscriptions*: Para la creación de *subscriptions* a *streams* se ha de de realizar una inserción mediante el SDK de Couchbase al *bucket* de *subscriptions* introduciendo: un nuevo UUID generado por el servidor y la definición de la *subscription* como cuerpo de la petición. Como una *subscription* se realiza sobre *streams*, debe corroborarse que son válidos. Debido al asíncronismo de Node.js, se usa la librería `async` para llamar a una función que cree la suscripción una vez haya terminado la verificación de los *streams*. Esto generará un documento en JSON en Couchbase con el siguiente formato (ver Listado 7.5):

```
1 {
2   "origin": "04324e09-2fac-41ca-bc79-ae8891a52ec2",
3   "destination": "912ece9d-8f0e-4040-bad1-478cfd899cd0",
4   "tenant":"fdd234-919f-4b682f6d--a1be477b-f66035"
5   "inputs": [
6     "B"
7   ]
8 }
```

Listado 7.5: Definición de una *subscription* en servIoTicy

- Consultar *subscriptions*: La consulta de *subscriptions* se realiza de la misma forma que la de los *streams*, hay que realizar una N1qlQuery a Couchbase. Concretamente una consulta que devuelva el documento (`meta(subscriptions).id`) en el que el campo *tenant* (propietario) sea el mismo que el identificador implícito en el token.
- Consultar *subscription*: La consulta de una *subscription* se realiza de la misma forma que la de las *subscriptions*, excepto por filtrar el resultado con el identificador de la *subscription* correspondiente.
- Modificar *subscription*: Para la modificación de suscripciones se utiliza el mismo método que en la creación excepto por la generación del identificador, que esta vez deberá ser el mismo que se ha recibido por URL. También es necesaria la verificación de los *streams* que forman la *subscription*, así que se utiliza la librería `async` del mismo modo.
- Eliminar *subscription*: La eliminación de una *subscription* es sencilla, basta con eliminar el documento de Couchbase con una llamada al SDK de Couchbase utilizando el identificador de la *subscription* a eliminar.

7.3.7. Rutas REST para la gestión de *Updates*

Para la implementación de la gestión de *updates* se ha lidiado además de, con Couchbase, con Kafka. Para realizar la conexión se ha utilizado el módulo *kafka-node*, una implementación de código libre disponible en los repositorios oficiales.

- Consultar *updates*: La consulta de updates está implementada mediante una N1qlQuery a Couchbase. Más concretamente, una consulta que devuelva el contenido de la última update con identificador igual al que forma parte de la URL (`meta(updates).id`).
- Insertar *updates*: Esta función se hace íntegramente utilizando el módulo de Kafka `kafka-node`. Para enviar datos a Kafka, se debe definir un Producer y la dirección IP de Kafka. El producer en este caso será la API REST. Para enviar los datos, primero se ha de generar un *KeyedMessage*, en que la *key* será el identificador del *stream* que forma parte de la URL y el contenido será un JSON como el del Listado 7.6. Esta estructura está definida con compatibilidad al software que recibirá los datos que Kafka envíe, el sistema de computación en tiempo real. No puede ser otra. Esta estructura debe ser rellenada con dos campos:
  - El identificador del *stream*.
  - `element`: El propio valor del *update*. Esta estructura es libre siempre y cuando cumpla el formato especificado por servIoTicy. Su estructura no afecta en ningún caso al diseño de la API, es decir puede cambiar sin ningún inconveniente.
  - Un *timestamp* del momento del envío.

```
1  {
2    'subscription': {
3          'destination': req.params.stream_id //Identificador del stream.
4    },
5    'update': {
6      'values': element, //El valor de la update.
7      'offset': {
8        'timestampMS': (new Date).getTime()
9    }
10 }
```

Listado 7.6: Contenido de un mensaje a Kafka en servIoTicy

Una vez definido el mensaje, se utiliza la función `send` para enviarlo. Con dicha función, pueden enviarse uno o varios mensajes, funcionalidad que se aprovecha para realizar micro-batching. Adicionalmente, se debe especificar el *topic* de Kafka que, en este caso, es `update` por decisión del equipo.

Una vez enviado a Kafka, se devuelve una confirmación al usuario y los datos dejan de ser gestionados por la API y, por tanto, su gestión ya no forma parte del proyecto. Kafka enviará los datos al sistema de computación en tiempo real que computará los cálculos definidos por los usuarios y enviará el resultado a Couchbase (ver Figura 7.4).

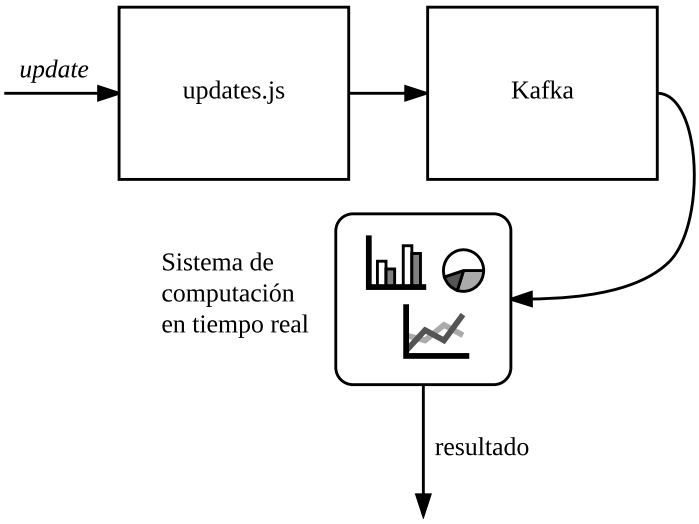


Figura 7.4: Envío de *updates* en servIoTicy.

7.3.8. Rutas REST para la gestión de Usuarios

Para la implementación de las rutas de gestión de usuarios se trabaja únicamente con MySQL, a través del módulo `mysql` para Node.js, un driver de código libre disponible en los repositorios oficiales.

- Crear *usuario*: Para la creación de usuarios se utiliza una query a MySQL que inserta en la tabla *Users* un identificador generado, el nombre de usuario, y el *hash* de la contraseña. Tanto el usuario como la contraseña se han de enviar como cuerpo de la petición a la API REST.
- Consultar *usuario*: Consultar el *username* de un usuario se realiza introduciendo su UUID cómo parte de la URL. Esta UUID se cotejará con la tabla Usuarios y devolverá el username del usuario.
- Eliminar *usuario*: La eliminación de los usuarios se realiza mediante una query que borra los usuarios que tengan el mismo UUID que el que forme parte de la URL.

7.3.9. Rutas REST para la autenticación de usuarios

- Autenticación: Para que un usuario se autentique contra la API es necesario enviar el usuario y la contraseña como cuerpo de la petición. Estos dos datos se cotejan contra la tabla de usuarios de MySQL que extraerá su UUID. Este UUID se encripta en un *token* y se envía de vuelta en la respuesta de la petición para que se utilice en las subsecuentes peticiones.

  Una descripción detallada de la autenticación puede verse en el apartado 7.3.11.

7.3.10. Middleware

En express.js las funciones *middleware* son aquellas que tienen acceso a la petición, a la respuesta y a la siguiente función dentro del ciclo petición-respuesta HTTP. Para la autenticación de la API ha sido necesario utilizar un módulo middleware para proteger las rutas únicas de cada usuario (ver figura 7.5).

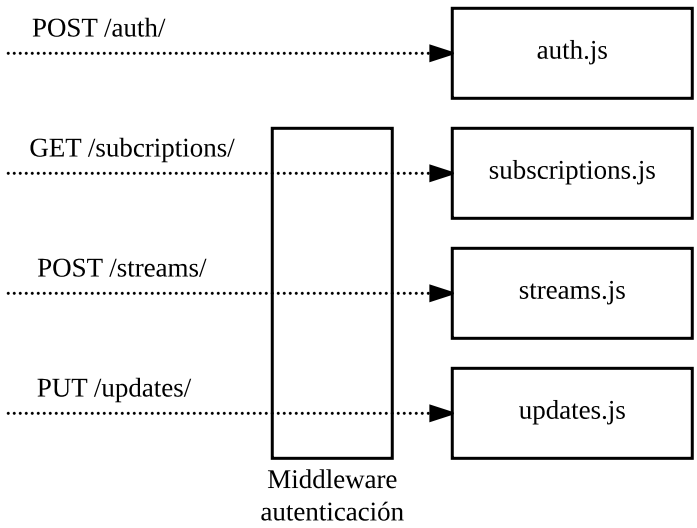


Figura 7.5: Rutas afectadas por el *middleware*.

Tal y como se muestra en la figura, las rutas de la API son distintas pero el *middleware* siempre es el mismo. Además, no todas las rutas deben pasar por el middleware de autenticación (si no, los usuarios no podrían autenticarse).

Una explicación más detallada del funcionamiento de este sistema se muestra justo con el sistema de autenticación en la sección 7.3.11.

7.3.11. Autenticación por JSON Web Tokens

La autenticación de usuarios se diseñó para usar *JSON Web tokens*. Para adoptar este sistema de autenticación por *token*, se ha utilizado el módulo *jsonwebtoken*, una implementación de *JSON Web Tokens* para Node.js de código abierto.

Esta implementación permite encriptar un objeto JSON (ver Listado 7.7) y decodificarlo a la vez que se verifica su validez y su caducidad (ver Listado 7.8). Para ello es necesario definir un *secret*, la clave con la que se encriptarán todos los *tokens* de la aplicación. Naturalmente, esta clave no debe ser revelada.

```
function() {
    //Incluir el módulo y definir el secreto
    var jwt = require('jsonwebtoken')
    var secret = "clave";
    \\Definir el objeto a encriptar y encriptar
    var payload = {"uuid":myuuid};
    var token = jwt.sign(payload, secret, {
        expiresIn: expiration
    });
    //Añadirlo a la respuesta
    res.json({
        token: token,
        expiresIn: expiration
    });
}
```

Listado 7.7: Encriptado de un objeto JSON con *jsonwebtoken*

```
function() {
    //Verificar el token
    jwt.verify(token, secret, function(err,decoded){
        if(err) {
            //No es válido, se deniega el acceso.
            console.log(err);
            return res.status(403).json({ApiError: "Invalid token"});
        }
        else {
            //Es válido, se deja pasar la petición.
            req.decoded = decoded;
            next();
        }
    });
}
```

Listado 7.8: Decodificación de un objeto JSON con *jsonwebtoken*

Para utilizar este sistema de autenticación ha sido necesario guardar información sobre los usuarios en la base de datos de autenticación. De momento solo ha sido necesario incluir una tabla para los nombres de usuario, sus identificadores únicos y sus contraseñas cifradas (ver Listado 7.9).

```
1 CREATE TABLE Users (
2       uuid varchar(255) NOT NULL PRIMARY KEY,
3       username varchar(255) NOT NULL UNIQUE,
4       password varchar(255) NOT NULL
5 );"
```

Listado 7.9: Tabla Users en MySQL

A continuación se detalla el funcionamiento del sistema de *login* de la aplicación. La numeración corresponde a la Figura 7.6

1. El dispositivo envía una petición de autenticación a la API a través de una ruta REST. Esta petición contiene el usuario y la contraseña, que llega a Auth.js, el módulo encargado de gestionar las rutas de autenticación.
2. El módulo verifica las credenciales contra la base de datos MySQL, que guarda los nombres de usuario y las contraseñas cifradas de todos los usuarios de servIoTicy.
3. Si las credenciales son correctas, MySQL devuelve como resultado el UUID del usuario.
4. El módulo recoge el UUID, lo codifica y lo manda como *token* de vuelta al usuario. A partir de aquí, todas las peticiones que haga el usuario deberán hacerse con el token como atributo durante el tiempo que éste tenga validez. Una vez caduque, deberá repetirse este proceso.

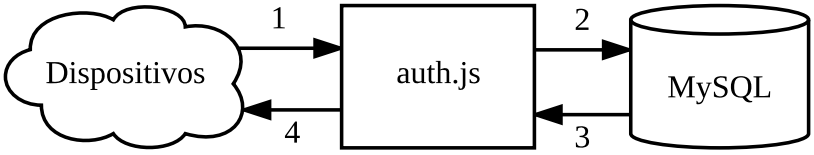


Figura 7.6: Sistema de autenticación.

Con el *token* generado, el usuario queda autenticado y puede realizar peticiones a URLs restringidas. El encargado de supervisar esta protección es el *middleware* de autenticación, un módulo que procesa todas las peticiones y verifica la validez del *token*. Si es válido, la petición se deja pasar, si no, se deniega el acceso.

A continuación se describe el funcionamiento del *middleware* de autenticación. La numeración corresponde a la Figura 7.7.

1. El dispositivo manda una petición a la API con el token como cabecera. El punto de entrada a la API es el *middleware* de autenticación.
2. El *middleware* procesa el *token*, lo verifica, decodifica el UUID y lo añade a la petición. A partir de este momento el UUID está implicito en la petición. La petición sigue su curso hacia el módulo de la ruta escogida, en este ejemplo, `/api/streams`. Gracias a que el UUID está

implicito, no hay necesidad de añadir otro campo a la *URL*, lo cual permite un diseño mucho más cercano al marco *REST*.

3. La petición se procesa en el módulo de la ruta. En este ejemplo, se utiliza el UUID para pedir a Couchbase los *streams* del usuario con ese identificador.
4. Couchbase devuelve los datos al módulo. En este punto se crea la respuesta a la petición que ya no contiene ningún *token*.
5. El módulo envía la respuesta al *middleware* que no efectua ninguna acción sobre ella.
6. El usuario recibe la respuesta a su petición.

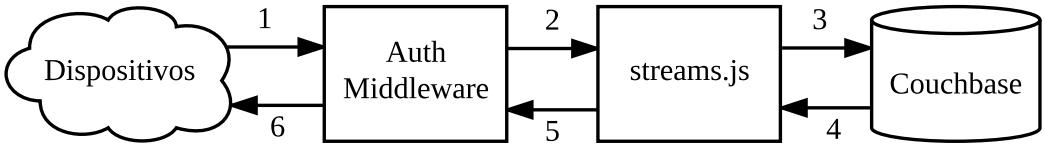


Figura 7.7: Funcionamiento del *middleware* de autenticación.

7.3.12. Pool de conexiones MySQL

La conexión con el servidor MySQL se realiza a través del *driver*[2] (controlador) *mysql* para Node.js, un módulo de código abierto en Javascript, disponible en los repositorios oficiales. Este driver permite realizar conexiones al servidor MySQL y realizar consultas a la base de datos, entre otras muchas funcionalidades que no se van a explotar en este proyecto.

El *driver mysql* permite crear conexiones a MySQL de dos maneras:

- Realizando conexiones únicas cada vez, creando un objeto por cada conexión. Esta conexión se realiza implícitamente en el momento que se ejecuta la *query* y se termina justo después. Se puede ver un ejemplo de esta implementación en el Listado 7.10.

[2]Programa informático que permite interaccionar con otro programa informático o dispositivo a través de una interfaz.

```
1  var mysql      = require('mysql');
2  var connection = mysql.createConnection({
3    host     : 'ejemplo.com',
4    user     : 'usuario',
5    password : 'contrasena'
6  });
7
8  connection.query('SELECT *', function (err, res, fields) {
9    if (error) throw error;
10   // Obtener resultados
11   connection.end();
12 });
```

Listado 7.10: Conexión simple con MySQL

En un principio se intenta hacer de esta forma hasta que el desarrollo se encuentra con un problema con esta implementación: si se corta la conexión con el servidor por un error, hace falta crear una conexión nueva. Al reconectar se crea otro objeto distinto pero las rutas siguen apuntando al objeto anterior (el de la conexión perdida). Se valora y se decide adoptar la conexión alternativa, explicada en el siguiente punto.

- Creando un *pool* de conexiones que permite reaprovecharlas una vez han acabado, reduciendo el *overhead* de creación y establecimiento de una nueva conexión. En este caso, se debe pedir una conexión al *pool* antes de hacer la *query*. Una vez hecha, las conexiones deben liberarse, no terminarse. Se puede ver un ejemplo simplificado de cómo se ha implementado esto en el Listado 7.11. Si una conexión termina por error, se elimina del *pool* y se crea una nueva (el driver hace esto automáticamente).

  Este planteamiento facilita la gestión de la conexión en el código, puesto que el *pool* de conexiones es siempre el mismo objeto y todas las rutas apuntarán a esta referencia. Se decide utilizar esta versión ya que gestionar independientemente cada conexión no ocasiona ningún beneficio para nuestra API y el diseño de la aplicación se vuelve más complejo.

```
var mysql = require('mysql');
var pool  = mysql.createPool({
  host     : 'ejemplo.com',
  user     : 'usuario',
  password : 'contrasena'
});

pool.getConnection(function(err, connection) {
  // Usar la conexión
  connection.query('SELECT *', function (error, results, fields) {
    // Obtener resultados y finalizar conexión
    connection.release();
    if (error) throw error;
  });
});
```

Listado 7.11: Pool de conexiones con MySQL

Las consultas se realizan en SQL adaptado al módulo, con los argumentos pasados por parámetro. De esta manera el módulo despliega de forma segura la *query* antes de mandarla a la base de datos y la protege de posibles inyecciones de código. Se puede ver una consulta de este tipo en el extracto de código de la API del Listado 7.12.

```
mySqlConnection.query('select uuid from `Users` where `username`=? and
`password`=?', [username,password], function(err,results,fields) {
    //Procesar la respuesta de la base de datos (variable results)
});
```

Listado 7.12: Consulta a MySQL

7.3.13. Conexión cifrada sobre TLS

Para la conexión cifrada de HTTP sobre TLS, se utiliza el módulo incluído en las librerías de Node.js `https`. Es necesario también incluir el módulo `fs` (*filesystem*) para poder acceder a los ficheros que contienen el certificado autofirmado (temporal) y la clave de cifrado.

Una vez cargados los dos ficheros, basta con realizar la llamada a `createServer` con los ficheros como parámetro y el puerto por el que se accederá a la API (ver Listado 7.13). En este caso es el 8000.

```
1  const https = require('https');
2  const fs = require('fs');
3
4  const options = {
5    key: fs.readFileSync('agent2-key.pem'),
6    cert: fs.readFileSync('agent2-cert.pem')
7  };
8
9  https.createServer(options, (req, res) => {
10   res.writeHead(200);
11   res.end('hello world\n');
12 }).listen(8000);
```

Listado 7.13: Ejecución de la API bajo HTTPS

Capítulo 8

Sistema de aprovisionamiento de servIoTicy sobre containers

A continuación se detallan los aspectos más relevantes sobre el segundo objetivo de este trabajo: la integración de un sistema de aprovisionamiento en containers para servIoTicy

8.1. Requisitos

El objetivo principal de esta fase es el aprovisionamiento automático de la infraestructura completa de servIoTicy. No obstante, el equipo ha decidido que se realice mediante el sistema de virtualización de containers, con los beneficios que comporta (ver Docker containers en Background de Tecnologías). Por tanto, la migración de los servicios de servIoTicy a containers es una tarea necesaria para poder desplegar la infraestructura completa con este sistema.

8.1.1. Migración de los servicios de servIoTicy a Docker *containers*

Actualmente, servIoTicy se despliega utilizando un sistema basado en máquinas virtuales, y se precisa que funcione sobre containers.

Los motivos de esta elección son varios: primero, por su flexibilidad. Los *containers* otorgan una flexibilidad al entorno que permite intercambiar *softwares* en cualquier momento de forma sencilla. También, al ser imágenes prediseñadas, se gana en velocidad de despliegue, siendo únicamente necesario la descarga de estas imágenes. Otro de los motivos también es que en el pasado han habido problemas con otros sistemas. Puppet, por ejemplo, cuenta con *scripts* de aprovisionamiento hechos por la comunidad para los *softwares* de servIoTicy, pero, o son versiones incompatibles o directamente ya no están mantenidos. El departamento encontró en Docker un recurso perfecto para solucionar varios problemas en uno: *software* oficial, en que las versiones de los softwares eran muy antiguas.

Para poder desarrollar un sistema de aprovisionamiento basado en *containers*, se necesitará orquestar un conjunto de *containers*, con uno (como mínimo) para cada *software*. Habrá que tener en cuenta los requisitos especiales de:

Storm: Storm está dividido en diferentes nodos que habrá que configurar correctamente, y buscar una buena implementación de ellos en *containers*. Estos servicios son:

- Nimbus: Tipo de nodo *master* del cluster de Storm.
- Supervisor: Tipo de nodo *slave* del cluste de Storm.
- UI: Interfaz gráfica del *cluster*.

Además, Storm deberá leer una topología empaquetada en un archivo `.jar` (desarrollada por el departamento). Esta topología, que se ejecutará sobre Storm, está desarrollada en Java y será necesario el uso de la versión propietaria de Oracle, puesto que el rendimiento es mejor en la versión privativa. No es aceptable para el departamento el uso de OpenJDK (versión libre de Java). Éste no es el único motivo por el que se requiere Java Oracle. La topología utiliza Nashorn, un intérprete de JavaScript necesario para ejecutar código de los clientes dentro de servIoTicy, y sólo se incluye en la solución privativa de Java.

Couchbase: Couchbase necesita que se definan los *bucket* y se generen los *Global Secondary Indexes* (vistos en el apartado de desarrollo de la API REST), necesarios para optimizar el rendimiento de las búsquedas en la base de datos. En conjunto, se trata de inicializar Couchbase para que funcione bajo los parámetros de servIoTicy.

MySQL: Para que la API REST pueda funcionar, se deberá inicializar MySQL con la base de datos y las tablas necesarias (requeridas por la API REST, como se ha visto en el apartado de su desarrollo).

El resto de *softwares* utilizados para el funcionamiento de servIoTicy no tiene ningún requisito especial. En conjunto, todos los softwares que deberán ser migrados a *containers* servIoTicy son:

- Storm
- Node.js
- Couchbase
- Kafka
- MySQL
- Zookeeper

Estos *containers* deberán estar interconectados entre sí para poder jecutar servIoTicy y se deberá diseñar una red interna para que puedan tener comunicación entre ellos (ver figura 8.1).

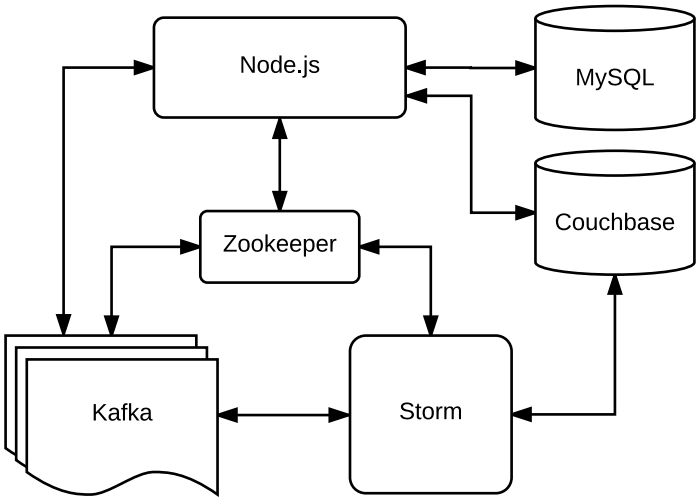


Figura 8.1: Arquitectura de los *softwares* de servIoTicy y sus conexiones.

Nota: en el apartado de desarrollo de la API REST, por simplicidad, se ha elidido el software Zookeeper debido a que es un elemento que sirve para mantener un control de estado de Kafka y Storm y no afecta en ningún aspecto al desarrollo de la API REST. En este apartado en cambio, es relevante tener en cuenta a Zookeeper en la infraestructura porque es un servicio importante sin el que servIoTicy no podría funcionar y forma parte del trabajo de final de grado conseguir que funcione en un entorno basado en *containers*.

8.1.2. Aprovisionamiento automático de *containers* con *Docker-compose*

Una vez se consiga tener la infraestructura ejecutándose en *containers*, se deberá conseguir un sistema de aprovisionamiento automático de la plataforma, es decir, una herramienta que permita crear la infraestructura desde cero en cualquier sistema operativo compatible con Docker y permita ejecutar servIoTicy sobre *containers*.

Para ello, se deberá diseñar una plantilla *Docker-compose* con los diferentes servicios. *Docker-compose* es el sistema de referencia de Docker para orquestar diferentes containers, y el equipo lo requiere por su flexibilidad, su comunidad, para poder ser mantenido fácilmente en el futuro y por su compatibilidad con Rancher.

Esto tendrá diferentes usos, y es que *docker-compose* puede ejecutarse por sí solo sin necesidad de un *dashboard* que lo gestione desde arriba. Simplemente basta con proporcionar el fichero de plantilla y ejecutar *docker-compose* con ella. Esto será usado por algunos desarrolladores para realizar pruebas de rendimiento y otros temas relacionados con distintas investigaciones. La solución completa debe integrar Rancher ya que, en un futuro, se pretendería ofrecer esta interfaz gráfica a los usuarios. De ahí que se deba cuidar la compatibilidad con Rancher y usar versiones compatibles.

8.1.3. Compatibilidad con Rancher

Rancher es un software propuesto por el equipo de desarrollo de servIoTicy para dotar de interfaz gráfica al sistema de aprovisionamiento diseñado para servIoTicy. Con Rancher, un usuario podría loguearse a la página web, servida desde los servidores del departamento, añadir el host sobre el que desee ejecutar servIoTicy y automáticamente se descargarán y ejecutarán los containers en la máquina añadida. Para ello, la plantilla de la infraestructura de servIoTicy, diseñada con el formato de *docker-compose*, deberá ser añadida y publicada a sus usuarios registrados para que los clientes puedan acceder a ella. Los usuarios también podrían añadir sus propias plantillas o modificar la proporcionada y desplegar sus servicios en sus máquinas. Rancher se encargará de descargar las imágenes de los *containers* desde *Docker Hub* y se comunicará con los distintos *hosts* para aprovisionar los distintos servicios (ver Figura 8.2).

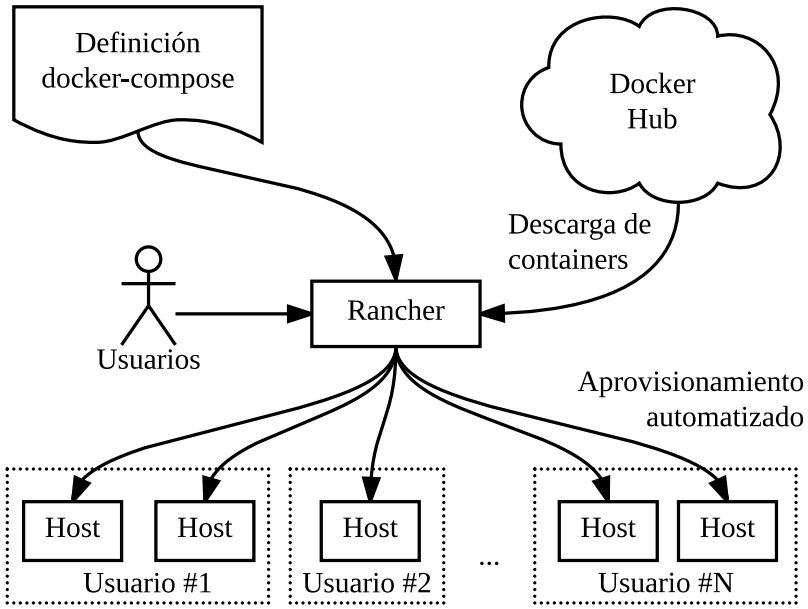


Figura 8.2: Diagrama del típico funcionamiento de Rancher con docker-compose.

Rancher es una herramienta muy potente, y ofrece muchas funcionalidades que no forman parte del trabajo de final de grado, pero es una herramienta con la que el TFG debe ser compatible para poder usar en el futuro tecnologías de escalabilidad de *containers* y orquestación de los servicios de su plataforma en nodos distribuidos y *clusters*.

8.1.4. Resumen de requisitos

En resumen, los requisitos que se extraen de estos contenidos son:

- Migrar los servicios de servIoTicy a Docker *containers*.

- Asegurar un funcionamiento correcto entre los distintos elementos de *software*.
- Realizar un aprovisionamiento de servIoTicy con *Docker-compose*.
- Asegurar la compatibilidad con Rancher para su reutilización en el futuro.

8.2. Diseño

A continuación se muestra el diseño de la solución proporcionada por el autor.

8.2.1. Arquitectura de los servicios de servIoTicy usando *containers*

Para poder desplegar servIoTicy en *containers* es necesario tener en cuenta los distintos servicios que se ejecutan. Como se ha visto en la especificación de requisitos préviamente, los servicios que se deberán ofrecer desde *containers* son:

- Nimbus (Storm)
- Supervisor (Storm)
- UI (Storm)
- Kafka
- Node.js
- MySQL
- Couchbase
- Zookeeper

Se propone que cada uno de los servicios se sirva desde un *container*, maximizando la flexibilidad que debería tener una infraestructura servida con este tipo de sistema de virtualización. De esta manera, se define el mínimo de *containers* necesarios para que servIoTicy funcione (ver figura 8.3).

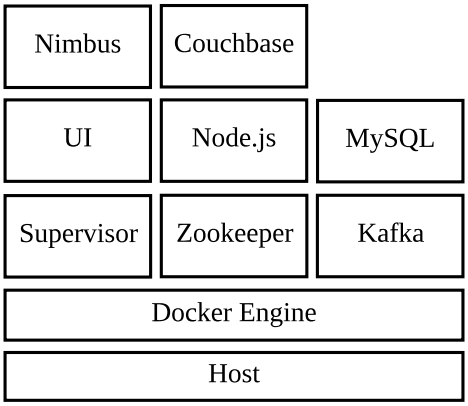


Figura 8.3: Arquitectura de servIoTicy en *containers*

Esta solución permitiría en un futuro escalar los recursos añadiendo más hosts y *containers*. En el caso de Node.js, se podría hacer un balance de carga entre API REST, y lo mismo con MySQL. Para

el resto, los softwares están preparados para poder trabajar en *cluster* (agrupados), por ejemplo, Nimbus detecta automáticamente todos los nodos Nimbus en la red. La dificultad de esta tarea reside en decidir qué proporción de *containers* es necesaria entre ellos. De todas formas, esta valoración no entra dentro del trabajo de final de grado, no obstante, forma parte del diseño que sea compatible con la filosofía de una infraestructura escalable.

8.2.2. *Containers* de soporte e inicialización

En el apartado anterior se muestra una estructura ideal de containers, no obstante, estos containers contendrán los servicios en un estado sin inicializar. Esto le otorga una flexibilidad muy importante al sistema (no hay que recrear los containers cada vez que haya un cambio en los parámetros de inicialización de los *software*) a cambio de realizar la inicialización de estos softwares mediante otros métodos. Siguiendo una filosofía de *containers* y microservicios, se podría crear un *container* temporal de soporte que efectúe los cambios necesarios, y luego fuera destruido.

Según los requisitos, esto sería necesario para Storm, Couchbase y MySQL, y cada uno realizando las inicializaciones necesarias para sus *containers* respectivos. Con esto, se debería ejecutar un container para Storm que cargara la topología (`.jar`), otro container para inicializar los buckets y los índices de Couchbase y otro para inicializar la base de datos de MySQL. En conjunto, así quedarían los softwares ejecutados e inicializados (ver Figura 8.4):

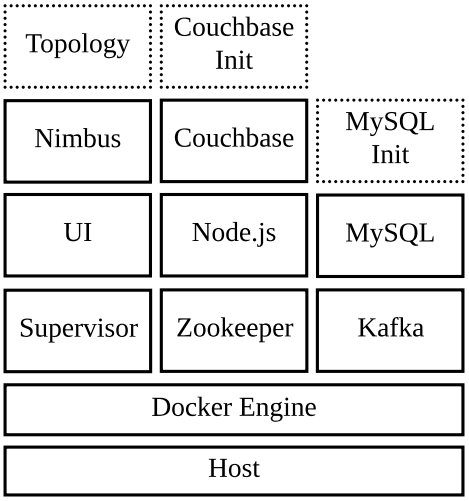


Figura 8.4: Arquitectura final de servIoTicy en *containers*, con los respectivos containers de soporte.

Una alternativa a este enfoque es la de recrear los *containers* aplicando las inicializaciones antes de su ejecución. Este método es completamente viable pero se ha descartado para no tener que realizar este paso en caso de cambiar los parámetros de inicialización en el futuro, simplificando las tareas a cambio de ejecutar un container temporal al principio, que no repercutirá en el funcionamiento una vez haya terminado sus funciones.

8.2.3. Mecanismos de resiliencia y control de orden de arranque

En sistemas distribuidos no se puede asumir que todos los servicios estén funcionando en todo momento. Si las máquinas pierden conectividad o simplemente sufren una parada inesperada del servicio, toda la infraestructura podría sufrir un error no controlado. Con *containers*, la filosofía debe ser la misma, puesto que los containers no dejan de ser versiones reducidas de sistemas operativos. Por este motivo se deben tener en cuenta mecanismos que controlen estos errores y permitan que se mantenga la máxima parte de la infraestructura en estado operativo. Algunos *softwares* diseñados para trabajar con sistemas distribuidos ya cuentan con este tipo de mecanismos, por ejemplo, Storm, que está preparado para trabajar en *cluster*. Sin embargo, algunos de los *software* que utiliza servIoTicy no están preparados para trabajar así, y habrá que adaptarlos. Otros, simplemente será necesario que esperen a ejecutarse en el momento adecuado, sobretodo los *containers* de soporte, que solo pueden ejecutarse en el momento en que los *containers* objetivo estén listos y desplegados. A continuación se describen los cambios que deberán hacerse para conseguirlo:

- API REST (Node.js): La API está conectada a Couchbase, MySQL y Kafka. Los módulos que implementan los clientes de Couchbase y Kafka ya cuentan con este tipo de control, y se puede recuperar la conectividad sin problema. En cambio, cuando se pierde conectividad con MySQL, se genera una excepción que interrumpe el proceso de Node.js. Para subsanarlo, se ha tenido que rediseñar la API REST para que funcione con un pool de conexiones. Así, las conexiones pueden perderse o recuperarse de forma transparente y sin necesidad de capturar errores específicos (ver seccion 7.3.12).
- MySQL Init: El *container* que inicialice la base de datos, ejecutará un *script* remoto contra el *container* de MySQL. Pero este *script* deberá ejecutarse cuando MySQL esté listo para recibir conexiones. Habrá que implementar un sistema que espere a que estén listos los servicios antes de inicializarlos.
- Couchbase Init: Couchbase debe inicializarse de la misma forma que MySQL, así que se deberá usar el mismo método para esperar a que esté listo antes de inicializarse.
- Topology (Storm): La inicialización de Storm para por cargar la topología de computación. Ésta sólo se podrá cargar a Storm cuando el cluster esté completamente listo para recibir conexiones. Se deberá usar el método ya mencionado y esperar a que los servicios se inicialicen.

8.3. Integración

En esta sección se integran sobre *Docker*, *Docker-compose* y *Rancher* las decisiones de diseño previamente explicadas.

8.3.1. Selección y adaptación de *containers*

Para realizar la migración de los *containers* se han escogido containers de los repositorios oficiales de Docker (Docker Hub). Se ha intentado elegir *containers* oficiales de cada organización, en la

medida de lo posible, para contar con soporte oficial. A continuación se detalla y justifica la elección del *container* para cada servicio y el proceso de adaptación para cumplir los requisitos.

- Nimbus, Supervisor y UI: El *container* oficial de *Storm* [1] ofrece una solución muy interesante. Un mismo *container* base cuenta con los tres servicios, pudiendo ejecutar el deseado en cualquier instancia. El único problema es que la imagen del *container* usa una instalación de *OpenJDK*. Si se recuerdan los requisitos, la versión libre de Java no es aceptable. Por ello, se ha tenido que modificar el Dockerfile de la versión oficial y utilizar una imagen base con *Java Oracle*.

  La imagen escogida es `anapsix/alpine-java`[2], un container base que incluye la versión privativa de Java, creado por Anapsix, un contribuyente de la comunidad. La elección está basada en la valoración que la comunidad hace sobre los containers dentro del propio *Docker Hub*. No obstante, esta versión elimina *Nashorn*, otro de los requisitos necesarios de la topología y hay que adaptarlo. Por tanto, estos son los pasos necesarios para la adaptación del container de *Storm*:

  1. Modificar el Dockerfile de `anapsix/alpine-java` para que no elimine `Nashorn`. Como se puede ver en el extracto del Dockerfile (ver Listado 8.1), está ejecutando un `rm -rf` sobre algunos binarios, incluyendo *Nashorn*. Adaptarlo es tan sencillo como eliminar esa línea del Dockerfile.

```
1  rm -rf /opt/jdk/jre/plugin \
2    /opt/jdk/jre/lib/ext/nashorn.jar \ //Eliminar esta línea.
3    /opt/jdk/jre/lib/oblique-fonts \
4    /opt/jdk/jre/lib/plugin.jar \
5    /tmp/* /var/cache/apk/* && \
```

Listado 8.1: Extracto del Dockerfile de `anapsix/alpine-java`

  2. Una vez modificado el Dockerfile, se construye un *container* temporal que pueda usarse para la recreación del container oficial de *Storm*. Para ello se debe ejecutar el siguiente comando:

     ```
     $ docker build -t container_temporal .
     ```

     Este comando construye un *container* usando el Dockerfile modificado. Este *container* es una imagen base Alpine que contiene*Java Oracle* y *Nashorn*, es decir, ya cumpliría los requisitos.

  3. Una vez se obtiene el container temporal, se procede a generar el nuevo *container* de *Storm* con la nueva base. Para ello hay que modificar el Dockerfile de `storm` y sustituir la línea:

     ```
     FROM openjdk:8-jre-alpine
     ```

[1] `https://hub.docker.com/_/storm/`
[2] `https://hub.docker.com/r/anapsix/alpine-java/`

por

```
FROM container_temporal
```

y posteriormente ejecutar el comando que construirá el container bajo otro identificador. En este caso, se ha utilizado el usuario del autor: `agsergi` para marcar el container.

```
$ docker build -t agsergi/storm .
```

Una vez creado el container, se publica a Docker Hub con el comando:

```
$ docker push agsergi/storm
```

A partir de este instante, el *container* de *Storm* corriendo sobre *Java Oracle* puede ser utilizado por cualquiera desde Docker Hub.

- Kafka: Kafka no cuenta con *container* oficial, es decir, Apache Kafka no soporta oficialmente el uso de *containers*. No obstante, la comunidad valora de forma muy superior al resto la solución que implementa el usuario `wurstmeister`[3]. En este caso no hace falta ninguna adaptación del *container* para ejecutar Kafka. Para instalarlo es tan sencillo como hacer `pull` desde Docker Hub con el comando:

```
$ docker pull wurstmeister/kafka
```

- Node.js. En este caso si se cuenta con una implementación y soporte oficial. El método que propone Node.js para su uso en *containers* es el siguiente:

  1. Crear un *container* nuevo partiendo del oficial[4]. Para ello hay que crear un *Dockerfile* con el siguiente contenido:

     ```
     FROM node:4-onbuild
     EXPOSE 8080
     ```

     donde la primera línea indica el *container* de base que se va a usar (el oficial) y la segunda el puerto que se va a exponer en la red.
  2. Una vez creado, es suficiente con construirlo y subirlo a Docker Hub con los comandos `build` y `push` como se ha visto en el apartado anterior. En este caso se ha identificado el container como `agsergi/rest`.

- MySQL, Couchbase y Zookeeper: Estos tres *software* cuentan con versión y soporte oficial de sus respectivas organizaciones y no requiere ningún tipo de adaptación. Para instalarlos es suficiente con utilizar el comando `pull` de los *containers* `mysql`[5], `couchbase`[6] y `zookeeper`[7].

[3] `https://hub.docker.com/r/wurstmeister/kafka/`
[4] `https://hub.docker.com/_/node/`
[5] `https://hub.docker.com/_/mysql/`
[6] `https://hub.docker.com/_/couchbase/`
[7] `https://hub.docker.com/_/zookeeper/`

8.3.2. Adaptación y funcionamiento de los *containers* de soporte e inicialización

Como se ha visto en la sección de diseño, la inicialización de los *containers* se hará a través de *containers* temporales de soporte. En este apartado se detalla exactamente el comportamiento de este tipo de *containers*.

- *Topology*: Este *container* es el mismo que el de los servicios de Storm (nimbus, supervisor y UI) pero en vez de ejecutar uno de esos servicios ejecutará el comando `storm jar` junto con la ruta de la topologia (otro .jar). Es necesario indicarle el *hostname* del *container* en el que está ejecutándose `nimbus` para que pueda añadir la topología al *cluster*. (El comando exacto se ve en más detalle más adelante en la sección de la integración con *docker-compose*).
- *MySQL init*: Este *container* es el mismo que el utilizado para servir MySQL. Sin embargo, lo que hará es ejecutar un *script* con los parámetros necesarios para inicializar la base de datos (ver Listado 8.2).

```
1 echo "CREATE TABLE Users ( uuid varchar(255) NOT NULL PRIMARY KEY,
2 username varchar(255) NOT NULL UNIQUE, password varchar(255) NOT NULL );"
3 | mysql -h mysql rapids -urapids -prapids
```

Listado 8.2: Código para inicializar la base de datos de MySQL

- *Couchbase Init:* Este *container* es el mismo que el de los servicios de Couchbase, pero al igual que `MySQL Init`, lo que hará es inicializarlo con un *script* que usa los binarios `couchbase-cli` y `cbq`, dos intérpretes de líneas de comando que vienen incluído en el container para poder automatizar la inicialización de los `bucket` y los *Global Secondary Indexes* (ver Listado 8.3).

```
1  [...]
2  /opt/couchbase/bin/couchbase-cli cluster-init \
3      -c couchbase:8091 \
4      --user=admin \
5      --password=password \
6      --cluster-init-username=admin \
7      --cluster-init-password=password \
8      --cluster-init-ramsize=1200 \
9      --service='data;index;query' \
10     --wait
11 [...]
12 /opt/couchbase/bin/cbq -engine=http://couchbase:8093 <<'EOF'
13 CREATE PRIMARY INDEX ON `subscriptions` USING GSI;
14 CREATE INDEX subscriptions_tenant_index ON `subscriptions`(tenant) USING GSI;
15 [...]
```

Listado 8.3: Extracto de código para inicializar la base de datos de Couchbase

8.3.3. Mecanismos de resiliencia y control de orden de arranque

Tal y como se ha explicado en el apartado de diseño, la mayoría de software de servIoTicy están preparados para afrontar adversidades en la conexión. No era el caso de Node.js y MySQL, que se tuvieron que rediseñar, como se ha visto en su correspondiente apartado. No obstante, sí que ha habido que realizar una importante integración con respecto al orden de arranque de los containers.

Como se ha comentado previamente, los *containers* de soporte deben esperar a la inicialización de aquellos a los que van a inicializar. Este mecanismo requiere que un *container* espere a que el otro esté completamente inicializado, y su servicio completamente operativo. No es suficiente con esperar a que el *container* haya terminado de iniciarse, puesto que un servicio puede tardar mucho más en arrancar que el propio *container*.

Para llevar a cabo esta tarea, se ha utilizado un sistema que espera a que un determinado puerto responda de forma correcta a una llamada remota. *Docker* ya recomienda el uso de estos mecanismos, y es en su página web donde directamente recomiendan el uso del *script* `wait-for-it.sh` [20].

Este *script* está implementado por el usuario `vishnubob` y su implementación es completamente pública. [8]. Su funcionamiento es muy sencillo, espera a que un puerto responda correctamente antes de ejecutar un comando (ver Listado 8.4).

```
1 $ ./wait-for-it.sh -t 0 www.google.com:80 -- echo "google is up"
2 wait-for-it.sh: waiting for www.google.com:80 without a timeout
3 wait-for-it.sh: www.google.com:80 is available after 0 seconds
4 google is up
5
6 $ ./wait-for-it.sh www.google.com:81 --timeout=1 --strict -- echo "google is up"
7 wait-for-it.sh: waiting 1 seconds for www.google.com:81
8 wait-for-it.sh: timeout occurred after waiting 1 seconds for www.google.com:81
9 wait-for-it.sh: strict mode, refusing to execute subprocess
```

Listado 8.4: Ejemplos de funcionamiento del *script* `wait-for-it.sh`

La utilización de este *script* se ve en más detalle en la integración con *Docker-compose* para cada container que lo utiliza.

8.3.4. Definición de los servicios en *Docker-compose*

Una vez todos los servicios de servIoTicy se han migrado a *containers* se debe orquestar su funcionamiento mediante un fichero *docker-compose*. *Docker-compose* utiliza ficheros `.yml` para definir las plantillas, y son compatibles con Rancher. A continuación se muestra cada parte relevante de la plantilla junto al servicio que representa y los distintos detalles de cada configuración:

- Nimbus (Storm): Para realizar la definición de *nimbus* en el *docker-compose.yml* basta con

[8] `https://github.com/vishnubob/wait-for-it/blob/master/wait-for-it.sh`

añadir los puertos de la aplicación, bajo la directiva `ports` para que se pueda acceder desde fuera de la red de *containers* y el comando a ejecutar, que en este caso es `storm nimbus`, con una ristra de parámetros proporcionados por el departamento. La ristra de parámetros incluye algunas especificiaciones de *hardware* y de *software* para la topología. Obsérvese como el container utilizado no es el oficial de *storm* si no la imagen modificada por el autor para cumplir con los requisitos de servIoTicy. Es la siguiente (ver Listado 8.5):

```
1  nimbus:
2      image: agsergi/storm
3      hostname: nimbus
4      ports:
5      - 6627:6627/tcp
6      command:
7      - storm
8      - nimbus
9      - -c
10     - worker.heap.memory.mb=4096
11     - -c
12     - topology.worker.childopts=-XX:+UseG1GC -Xms4096m
13         -XX:MaxGCPauseMillis=20 -XX:SurvivorRatio=12
14         -XX:InitiatingHeapOccupancyPercent=35 -XX:+DisableExplicitGC
```

Listado 8.5: Definición en *docker-compose* del servicio de *Storm*: `nimbus`

- Supervisor (Storm): En la definición de *supervisor* no es necesario exponer puertos y basta con definir el comando a ejecutar, en este caso `storm supervisor` (ver Listado 8.6).

```
1  supervisor:
2      image: agsergi/storm
3      hostname: supervisor
4      command:
5      - storm
6      - supervisor
```

Listado 8.6: Definición en *docker-compose* del servicio de *Storm*: `supervisor`.

- UI (Storm): UI es el servicio que ofrece una interfaz gráfica para Storm. Se debe exponer el puerto de la aplicación para que se pueda acceder desde fuera de la red de *containers*, y así poder ver lo que se está ejecutando dentro de *Storm*. En este caso, se sirve desde el puerto 8080 y se redirige hacia otro distinto (8081 pero podría ser otro) ya que en la máquina en que se ejecuta ya hay otro servicio en el 8080 (ver Listado 8.7).

```
1    nimbusui:
2      image: agsergi/storm
3      hostname: nimbusui
4      ports:
5      - 8081:8080/tcp
6      command:
7      - storm
8      - ui
```

Listado 8.7: Definición en *docker-compose* del servicio de *Storm*: `UI`.

- Topology (Storm): Este container de soporte envía la topología al cluster. Para ello, se deberá esperar a que esté completamente inicializado. Aquí se puede ver por primera vez el uso del *script* `wait-for-it` esperando al puerto 6627 de *nimbus*, definidio en los apartados anteriores. Concretamente se espera indefinidamente y sin escribir ningun mensaje por pantalla (`-t 0 -s`). Cuando consigue contactar con *nimbus*, se ejecuta storm jar, que ya está preparado para enviar una topología al cluster. Esta topología, no obstante, debe sacarse de un directorio de la máquina en que se ejecuta. Para mapear estos directorios debe usarse la directiva `Volumes`, con la que se monta un directorio externo al container, dentro del container, consiguiendo así acceder a la topología proporcionada por el departamento. En este caso se ha hecho servir la carpeta de trabajo del autor. La definición es la siguiente (ver Listado 8.8):

```
1  topology:
2      image: agsergi/storm
3      hostname: topology
4      working_dir: /var/rapids
5      volumes:
6      - /home/agsergi/workspace/rapids-provisioning/docker:/var/rapids
7      command:
8      - ./wait-for-it.sh nimbus:6627 -t '0' -s --
9      - storm jar
10           /var/rapids/rapids-topology-0.1.0-SNAPSHOT-jar-with-dependencies.jar
11           es.bsc.rapids.Topology
12           -c
13           /var/rapids/rapids.properties
14           -t
15           topology
```

Listado 8.8: Definición en *docker-compose* del servicio de soporte de *Storm*: `Topology`.

- Kafka: El *container* de Kafka tiene unos requisitos especiales. A parte de definir los puertos (que se deben exponer ya que su desarrollador lo diseñó para que pudiera accederse a él únicamente desde el exterior) y las carpetas compartidas (que se realizan de la misma forma que en los apartados anteriores), se deben configurar una serie de variables de entorno para que funcione. Estas variables de entorno corresponden al *hostname* del container que servirá *Kafka*, el puerto por el que se servirá, el *topic* a crear (etiqueta para agrupar todos los mensajes provinientes de

la API) y la dirección y puerto de *zookeeper*, necesario para el control de estados (ver Listado 8.9).

```
1  kafka:
2    image: wurstmeister/kafka:0.9.0.1
3    hostname: kafka
4    environment:
5      KAFKA_ADVERTISED_HOST_NAME: kafka
6      KAFKA_ADVERTISED_PORT: '9092'
7      KAFKA_CREATE_TOPICS: updates:1:1
8      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
9    ports:
10   - 9092:9092/tcp
```

Listado 8.9: Definición en *docker-compose* del servicio de *Kafka*

- Node.js: Para Node.js simplemente es necesario exponer los puertos del servicio. Para ello se mapea el 8082 ya que el 8000 es utilizada por otra aplicación externa. Se puede apreciar como el container utilizado es el que se ha creado expresamente para servir la aplicación del autor (ver Listado 8.10).

```
1  rapidsrest:
2    image: agsergi/rest
3    hostname: rapidsrest
4    ports:
5    - 8082:8000/tcp
```

Listado 8.10: Definición en *docker-compose* del servicio de Node.js (API REST).

- MySQL. Para este *container* también hay que definir una serie de variables de entorno con la que se consigue preinicializar la base de datos. Básicamente, estas variables de entorno corresponden a la contraseña de *root*, el nombre de la base de datos a crear, y un usuario y contraseña para poder acceder (ver Listado 8.11). Naturalmente, estos datos deben tratarse con extrema precaución, aunque no debería haber peligro porque el fichero *docker-compose* nunca se debería poder ver desde el exterior. Nótese que no es necesario exponer ningún puerto ya que no se va a acceder al *container* desde fuera de la red de *containers*.

```
1  mysql:
2    image: mysql
3    hostname: mysql
4    environment:
5      MYSQL_ROOT_PASSWORD: passroot
6      MYSQL_DATABASE: rapids
7      MYSQL_USER: user
8      MYSQL_PASSWORD: pass
```

Listado 8.11: Definición en *docker-compose* del servicio MySQL

- MySQL Init: La inicialización de MySQL se lleva a cabo con su *container* de soporte. Se puede apreciar otra vez el uso del script *wait-for-it*. En este caso se ejecuta el *script* mencionado en la sección anterior, que inicializa las tablas de la base de datos. Para acceder al *script* desde el container se han de montar una vez más las carpetas del *host* de la misma forma que ya se ha comentado en apartados anteriores (ver Listado 8.12).

```
1  mysqlinit:
2    image: mysql
3    working_dir: /var/rapids/scripts
4    volumes:
5    - /home/agsergi/workspace/rapids-provisioning/docker:/var/rapids/scripts
6    command:
7    - ./wait-for-it.sh mysql:3306 -t '0' -s --
8        /var/rapids/scripts/mysql_init.sh
```

Listado 8.12: Definición en *docker-compose* del servicio de inicialización de MySQL

- Couchbase: Se requiere únicamente exponer los puertos necesarios para poder acceder a ellos desde fuera. Esta exposición podría ser elidida completamente si no fuera porque a veces se precisa poder ver el rendimiento de los nodos de *couchbase* desde su interfaz gráfica, a la que se accede mediante estos puertos (ver Listado 8.13).

```
1  couchbase:
2    image: couchbase:4.1.1
3    ports:
4    - 8091:8091/tcp
5    - 8092:8092/tcp
6    - 8093:8093/tcp
7    - 8094:8094/tcp
8    - 11210:11210/tcp
```

Listado 8.13: Definición en *docker-compose* del servicio de Couchbase

- Couchbase Init: Al igual que MySQL Init, se ejecuta un *script* desde una carpeta local que

hay que compartir con el *container*. Con `working-dir` se define el directorio desde el que se ejecutará (ver Listado 8.14).

```
1  couchbase-init:
2    image: couchbase:4.1.1
3    working_dir: /var/rapids/scripts
4    volumes:
5    - /home/agsergi/workspace/rapids-provisioning/docker:/var/rapids/scripts
6    command:
7    - sh
8    - /var/rapids/scripts/couchbase_init.sh
```

Listado 8.14: Definición en *docker-compose* del servicio de inicialización de Couchbase

- Zookeeper: Zookeeper requiere únicamente que se expongan los puertos. Esto no sería necesario si Kafka no requiriése que esté expuesto por razones de diseño de su desarrollador. Se puede observar que la imagen es la oficial (ver Listado 8.15).

```
1  zookeeper:
2    image: zookeeper
3    hostname: zookeeper
4    ports:
5    - 2181:2181/tcp
```

Listado 8.15: Definición en *docker-compose* del servicio de Zookeeper

8.3.5. Integración de la solución sobre Rancher

Ahora que ya se dispone de un fichero *docker-compose.yml* que define la infraestructura con *containers*, ya se puede utilizar para desplegar servIoTicy. De hecho, el departamento lo utilizará sin la interfaz gráfica para muchos de sus proyectos de investigación. No obstante, el departamento tiene la idea de productivizar servIoTicy y para ello se necesita una interfaz gráfica que facilite el consumo a clientes no expertos. Ahí es donde entra Rancher, la última plataforma que se integrará para esta solución.

La idea se basa en la adaptación de la plantilla *docker-compose* ya creada, que es compatible con Rancher [21]. Rancher permite su importación para crear las denominadas *Stacks*, las plantillas que podrán ser desplegadas en los *hosts* que los usuarios añadan.

Para que los usuarios puedan desplegar servIoTicy en sus máquinas se ha creado un *Stack* importanto el archivo *docker-compose.yml* No obstante, la importación no es satisfactoria sin un nivel de adaptación de formato previo. Algunos elementos descritos en el fichero *docker-compose.yml* no se importan de forma automática, ya que algunos de los comandos que se están definiendo (por ejemplo, los de Topology o Nimbus) son bastante complejos y muy sensibles a espacios, comillas, etc. Así que, para integrarlo completamente en Rancher se han tenido que realizar algunas pequeñas

adaptaciones que se pueden hacer directamente en la interfaz gráfica y con muy poco esfuerzo, puesto que no van más allá de un cambio de formato (ver Figura 8.5).

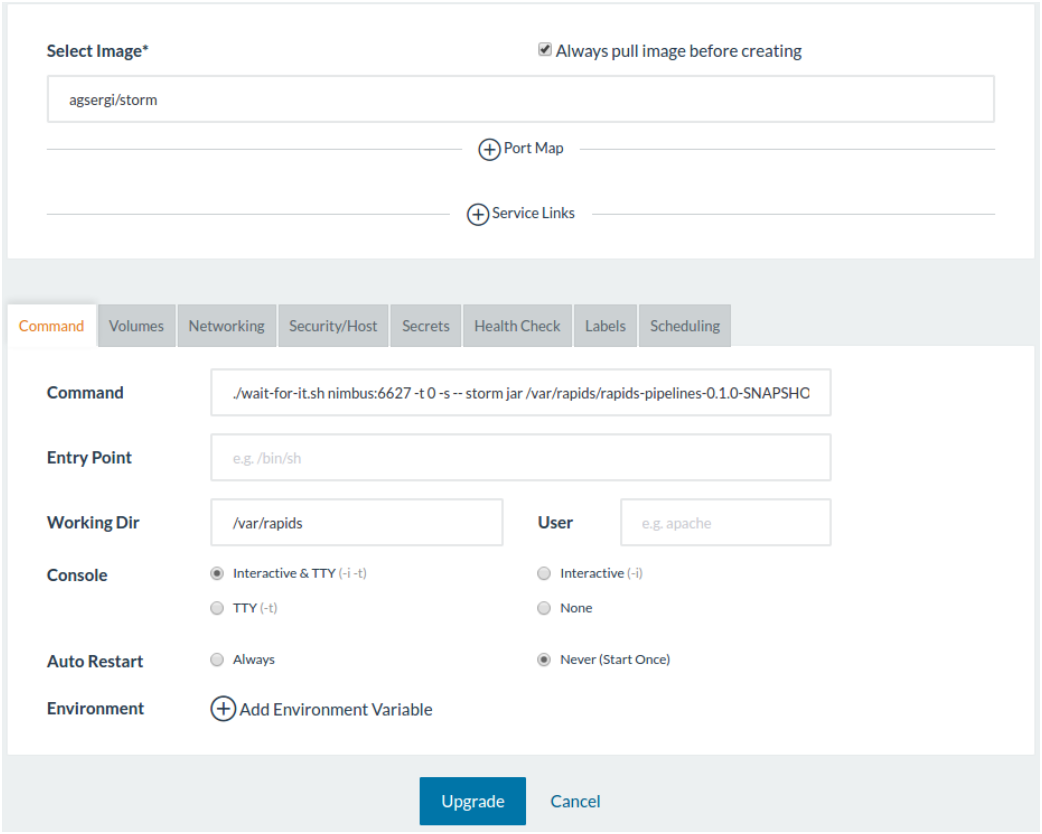


Figura 8.5: Captura de pantalla de la definición del servicio Nimbus en Rancher

Una vez creado el *Stack*, el usuario podrá encender, apagar y gestionar los diferentes servicios que están definidos dentro de la plantilla. (ver Figura 8.7).

Para poder probar la integración se ha añadido un *host* (el portátil de desarrollo del autor). Para añadir un *host*, basta con seguir el asistente que el propio *software* da (ver Figura 8.6). Se pueden añadir hosts de distintas clases e incluso de proveedores de *cloud* como *Amazon Web Services* o *Microsoft Azure*.

Figura 8.6: Captura de pantalla de la interfaz para añadir hosts en Rancher.

Por último, solo queda ejecutar toda la plataforma haciendo *click* en el símbolo de *play* (ver Figura 8.7).

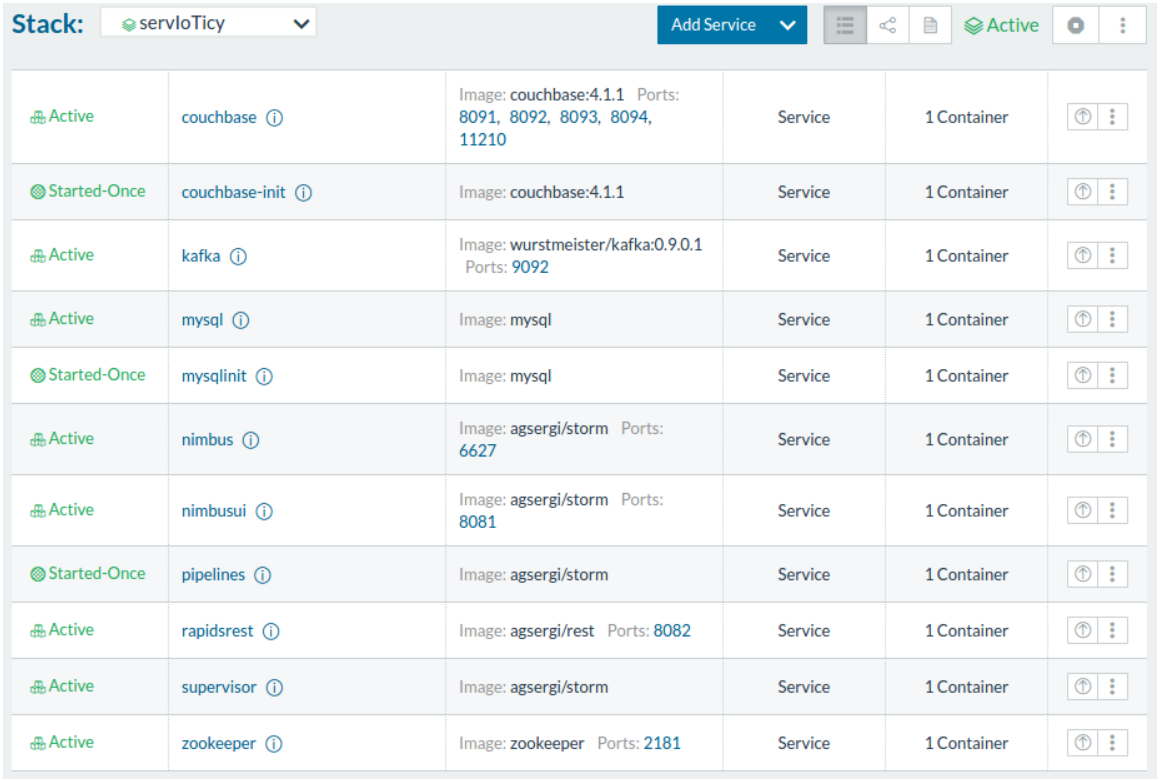


Figura 8.7: Captura de pantalla del *Stack* de servIoTicy ejecutándose en Rancher.

Así, se consigue satisfactoriamente el aprovisionamiento automatizado de servIoTicy en *containers*, finalizando la parte correspondiente al segundo objetivo de este trabajo de final de grado.

Capítulo 9

Análisis de rendimiento

En este capítulo se realiza una prueba del rendimiento que ofrece la API REST que se ha implementado durante el proyecto. Para ello se ha diseñado un experimento que reproduce el comportamiento que tendría un entorno de producción.

9.1. Objetivo

El objetivo de este experimento es observar el comportamiento de la API REST mostrando el número máximo de *updates* que puede soportar y la latencia que tienen las peticiones realizadas a distintos niveles de *micro-batching*.

9.2. Diseño del experimento

El experimento se basa en la extracción de *timestamps* (marcas de tiempo) mediante un *script* en Node.js que realiza peticiones a la API como si se tratara de un dispositivo.

La plataforma en la que se ejecuta el experimento es un portátil Dell Latitude E7450, que cuenta con un procesador Core i5-5300U y 8GB de RAM DDR3L. (**Nota**: El experimento solamente se ejecuta sobre un núcleo del procesador). El sistema operativo usado es Ubuntu 16.04.2 LTS, con la versión de núcleo Linux 4.4.0-72generic. La versión de Node.js usada es en el servidor 6.10.2 y 7.7.2 en el cliente, la versión de Kafka es la 0.9.0.1, y la plataforma corre sobre Docker Engine 1.12.

Para poder valorar el comportamiento, se somete a la API a diferentes niveles de estrés. Para ello se envían ráfagas de 10.000 (diez mil) peticiones separadas por un mismo *think time* (tiempo entre envío peticiones) que cambia a cada ráfaga, empezando en 15 milisegundos y reduciendo de uno en uno hasta 0 milisegundos (sin *think time*). Se repite el experimento para distintos niveles de *micro-batching*. Los niveles de *micro-batching* son los siguientes:

- Nivel de *micro-batching* 1: Conjunto de 1 *Update* (22 bytes)
- Nivel de *micro-batching* 5: Conjunto de 5 *Updates* (110 bytes)
- Nivel de *micro-batching* 10: Conjunto de 10 *Updates* (220 bytes)
- Nivel de *micro-batching* 15: Conjunto de 15 *Updates* (330 bytes)
- Nivel de *micro-batching* 20: Conjunto de 20 *Updates* (440 bytes)
- Nivel de *micro-batching* 25: Conjunto de 25 *Updates* (550 bytes)
- Nivel de *micro-batching* 30: Conjunto de 30 *Updates* (660 bytes)

La extracción de los *timestamps* se realiza en dos momentos distintos para poder calcular el tiempo de *Round Trip* (ida y vuelta):

- Justo antes del envío de la petición.
- Justo después de recibir la respuesta.

Por tanto, se está midiendo el tiempo que tarda la *API* en leer una petición, gestionar el contenido, enviarlo a Kafka, y enviar la confirmación al cliente. Como las pruebas se realizan en una misma máquina a través de las interfaces de *loopback*, el tiempo de transmisión de la información será ínfimo, y por tanto, la aproximación que se realiza al leer los tiempos desde el cliente en vez de hacerlo desde el servidor es razonable. Las medidas se han llevado a cabo de esta manera para simplificar las tareas, puesto que instrumentar el servidor es más complicado.

Además de extraer las marcas de tiempo, también se marca si la petición ha sido errónea o no y el *think time* con el que se hizo cada petición.

Una vez extraídos los datos, se realiza el cálculo del *throughput* usando la media aritmética de los valores observados, agrupando por décima de segundo y eliminando los 1000 primeros y últimos resultados (esto se hace ya que los valores del principio y del final se extraen en momentos en que la API aún no está saturada, y en momentos en que está muy saturada, respectivamente, pudiendo influir negativamente en la dispersión de los resultados).

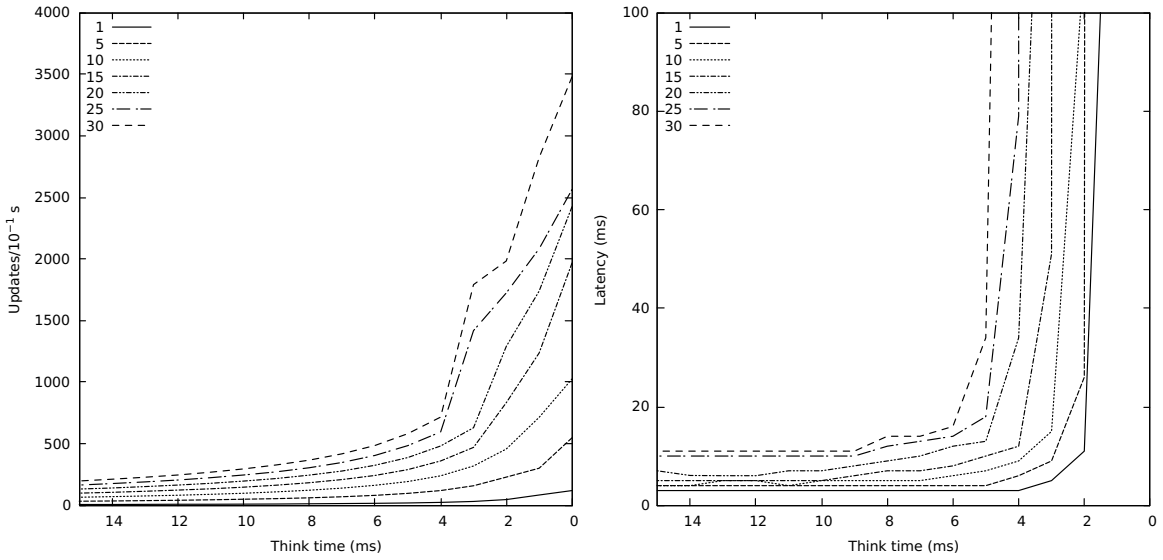
También se realiza el cálculo de la latencia usando el percentil 95 % para observar los peores casos de latencia en cada momento, valor más interesante que la media ya que su valoración debe contemplarse la calidad de servicio de un hipotético conjunto de usuarios (la media daria un valor de latencia inferior, pero escondería los peores resultados, y este análisis quiere contemplarse el caso peor).

9.3. Discusión de resultados

9.3.1. Micro-batching

Un aspecto que se observa claramente en los experimentos es el efecto del *micro-batching* en la mejora del *throughput*. Esto es observable en la curva de batching 30: si se selecciona el punto de mayor *throughput* con una latencia y dispersión de puntos razonable (ver figura 9.1), se ve que para un *think time* de 5 ms, el throughput es unas 30 veces superior (581 updates/ds) a su correspondiente valor sin batching (19 updates/ds), pero la latencia es solamente 11,3 veces superior (34 ms respecto a 3ms).

A simple vista puede parecer una mejora muy sustancial, pero sin embargo, esta comparación es injusta, puesto que el *think time* está limitando el número de peticiones que puede realizar la API sin batching (el *think time* se podría considerar como el tiempo necesario para agrupar 30 elementos antes de enviarlos, y no tiene sentido compararlo así, puesto que una ejecución sin batching no tendría que esperar tanto tiempo). Para ver que realmente el *micro-batching* es efectivo, se debe buscar el punto de mayor throughput con una latencia razonable pero en los dos casos: con y sin micro-batching e independientemente del *think time*.



(a) *Updates* por décima de segundo, para distintos niveles de *batching* y *think time*.

(b) Latencia por décima de segundo, para distintos niveles de *batching* y *think time*.

Figura 9.1: *Updates* por décima de segundo y latencia para distintos niveles de *batching* y *think time*.

Bajo estos requisitos se encuentra que, para la curva de *batching* 1 y un *think time* de 2ms (generar updates con una frecuencia superior genera latencias altas con una dispersión muy elevada), se genera un *throughput* de 46 *updates*/ds con una latencia de 11 ms. Comparando estos valores con los de batching 30 mencionados anteriormente, se observa que realizar un *micro-batching* de 30 *Updates*

genera un throughput 12 veces superior (581 *updates*/ds respecto 46 *updates*/ds) pero una latencia únicamente 3 veces superior (34 ms respecto 11 ms). Por tanto, comparando los dos mejores casos con y sin *micro-batching* se observa una clara ventaja en el uso de *micro-batching* con el objetivo de mejorar el *throughput* sin ganar demasiada latencia. Esto se debe a que se generan menos cabeceras HTTP por cada *Update*, reduciendo el gasto necesario para gestionarlas, consiguiendo así enviar más *Updates*

Cabe mencionar que a partir de un *batching* de 35 *Updates*, los resultados tienen una dispersión muy elevada y el servidor empieza a comportarse de forma errática, dando lugar a valores confusos que podrían llevar a conclusiones equivocadas. Por ese motivo se ha decidido limitar las gráficas y la discusión a un *batching* de 30 *Updates*.

Esto no significa necesariamente que un *batching* de 30 sea el nivel óptimo para la aplicación, puesto que el rendimiento que dé puede verse afectado por diversos factores. Hay que tener en cuenta que el uso de *micro-batching* es un balance entre el aumento del *throughput* y el de la latencia, y se debe sacrificar uno para tener el otro. Véase en la figura 9.1b, por ejemplo, que es imposible tener latencias inferiores a 10 ms con *batching* de 30 *Updates*, y en cambio, sí es posible con *batching* más pequeños, incluso con *think times* bajos. Además, dependiendo de la infraestructura que soporte el software en producción y de la latencia deseada, podría ser interesante intercambiar este valor o incluso cambiarlo dinámicamente según la carga del sistema. El nivel óptimo de *batching* y su *throughput* dependerán, por tanto, de la latencia que se desee garantizar a los clientes que envíen datos a servIoTicy. Además, cabe recordar que estos resultados son extraídos de una ejecución en un portátil y en un solo núcleo, así que las tendencias observadas mejorarían claramente en el uso de *hardware* adecuado.

9.3.2. *Throughput* y latencia

En líneas más generales, los resultados muestran una tendencia muy clara en cuanto a la relación entre el *throughput* y la latencia de las peticiones ya comentada en el párrafo anterior. Cuanto más *Updates* por décima de segundo realiza el sistema, más tardan en servirse las peticiones, puesto que el servidor está bajo una carga más elevada y éstas se encolan indefinidamente hasta que se gestionan. Cuando el servidor se satura (mucho *throughput*), las latencias se disparan de forma exponencial. Este comportamiento puede observarse en la figura 9.2 donde se ve que, independientemente del *batching*, si se realizan updates a muy alta frecuencia (sin *think time*) la latencia es varios órdenes de magnitud superior a lo que se podría considerar razonable en procesado de datos en tiempo real (alrededor de una centésima de segundo). Naturalmente, el nivel de *batching* afecta desplazando hacia la izquierda el punto a partir del cual se disparan las latencias, causado por un incremento en el número de bytes que la API y Kafka deben procesar.

Con este ejercicio, que incluye la realización de experimentos, tratamiento de los datos obtenidos y su posterior valoración, se cumple el tercer y último objetivo de este trabajo de final de grado.

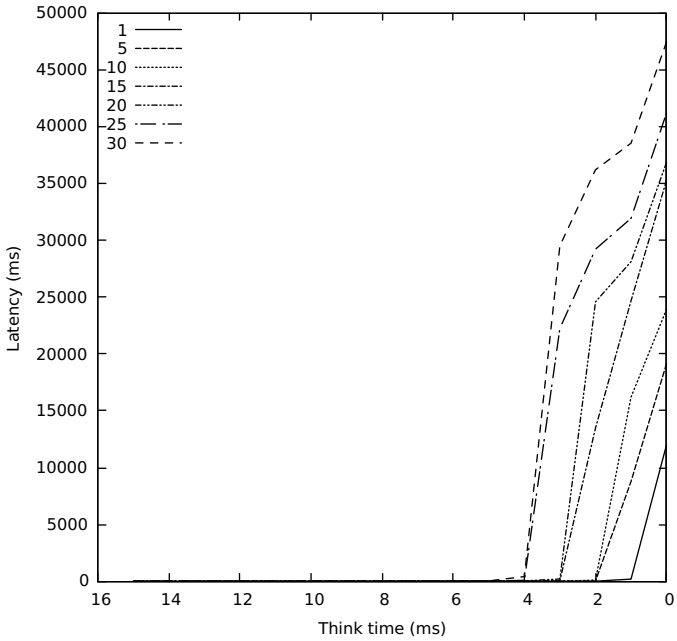


Figura 9.2: Latencia por décima de segundo, para distintos niveles de *batching* y *think time*. Se muestran todos los valores obtenidos.

Capítulo 10

Conclusiones y trabajo futuro

En este capítulo se revisan los objetivos cumplidos durante el trabajo y se comenta el trabajo futuro a llevar a cabo en relación a la solución propuesta.

10.1. Contribuciones de este trabajo

Durante este trabajo de final de grado se han proporcionado soluciones a problemas reales del proyecto servIoTicy, la plataforma de *stream processing* (procesado de flujos de información) desarrollada por el departamento Data-centric Computing del BSC. ServIoTicy requería una actualización de la plataforma para adaptarse a los nuevos tiempos y contar con herramientas que facilitasen su acceso a diferentes tipos de clientes. Para conseguir este propósito se definieron distintos objetivos para que se llevaran a cabo como parte del trabajo de final de grado del autor, actual trabajador del departamento.

Dentro de las tareas para realizar la modernización de servIoTicy se encontraba la renovación de la API REST. Esta tarea ha comportado la integración y desarrollo sobre diversas tecnologías, entre las que se destaca Node.js, que utiliza un paradigma de programación completamente distinto a los modelos tradicionales y ha requerido un esfuerzo adicional. La solución proporcionada permite que la plataforma servIoTicy pueda ejecutarse sobre un paradigma de programación completamente distinto al de la versión anterior, algo que repercute directamente en el rendimiento de la aplicación. Entre otras tecnologías que se han integrado en la API REST se encuentra también Couchbase, una base de datos NoSQL que, aunque ya se utilizara en el proyecto, integrarla en la API ha requerido también un esfuerzo extra debido a que no es el modelo tradicional al que el autor estaba acostumbrado. También se han integrado otras tecnologías como Storm, Kafka y Zookeeper, todas ellas tecnologías del estado del arte del *stream processing* y han sido la primera toma de contacto con los sistemas distribuidos del autor, sentando las bases del trabajo que se llevará a cabo dentro del departamento después de este trabajo.

La renovación de la API REST no ha sido la única tarea llevada a cabo para la actualización de servIoTicy. De nada sirve completar la infraestructura de servIoTicy si no se dan los mecanismos de

aprovisionamiento adecuados. Así, se estableció como objetivo proporcionar un sistema que desplegara el *software* directamente en el *hardware* de los clientes, cosa que ofrecería flexibilidad tanto al departamento que no tiene que costear los recursos, como al cliente que recibe la libertad necesaria para escalar los servicios bajo su propio control. La solución proporcionada no solo proporciona un sistema automatizado de aprovisionamiento, si no que una vez más acerca el proyecto al estado del arte gracias a la integración de los *Docker containers*, una solución de virtualización que rompe con los estándares tradicionales sobre el campo.

Por último, como ejercicio de evaluación de todo el trabajo realizado se han desarrollado una serie de experimentos que permiten ver el comportamiento de la API REST y que permiten cuantificar el rendimiento de la solución propuesta por el autor. Esto no repercute tanto en la mejora del propio proyecto de servIoTicy, si no en el hecho de poder contrastar la propuesta de este trabajo de final de grado con las alternativas que hay y que posiblemente haya en el futuro.

Así, tras haber cumplido los distintos objetivos antes mencionados, y tras haber integrado competencias técnicas propias de un ingeniero, incluyendo conocimientos adquiridos en la carrera y una importante parte de aprendizaje autónomo, se contribuye satisfactoriamente con este trabajo de final de grado a la plataforma servIoTicy, brindándole rendimiento, innovación y competitividad.

10.2. Trabajo futuro

En primer lugar, una de las lineas de continuación de este trabajo es la de la integración de mecanismos que mejoren la escalabilidad de servIoTicy y aumenten la disponibilidad. Se trabajará en integrar un sistema de balanceo de carga para la API REST, algo crítico para una solución en producción. Ahora mismo la API REST es el cuello de botella de la aplicación y es crítico ofrecer la capacidad de replicarla y balancear la carga entre ellas de forma sencilla para el usuario. Rancher ofrece diversos mecanismos para el reparto de carga, pero sería necesario adaptar la propia API REST.

En segundo lugar, se trabajará en la incorporación de alguno los orquestadores que integra Rancher como Kubernetes, Mesos o Swarm. Éstos ofrecerían un control a gran escala de servIoTicy pudiendo trabajar con grandes cantidades de nodos y desplegando los diferentes servicios según la demanda de la aplicación.

Otra linea de continuación de este trabajo es la de conectar un directorio activo como LDAP a Rancher para permitir que los usuarios se registren a través de una web y directamente tengan acceso al panel de control que Rancher proporciona. Así, Rancher quedaría finalmente integrado por completo en el producto final.

Por último, una linea de trabajo que no es crítica pero vale la pena mencionar es la del desarrollo de un *dashboard* en el que se visualicen los datos y se gestionen los dispositivos registrados en la plataforma en tiempo real. Este *dashboard* irá conectado a la API REST ya desarrollada y facilitaría la gestión a los clientes de la infraestructura, algo que le daría mucha visibilidad al producto y lo haría muy competitivo.

Esta es la hoja de ruta que seguirá el equipo de servIoTicy con el objetivo de crear una solución de *stream processing* robusta, innovadora, eficiente y competitiva.

Bibliografía

[1] J. Research. (2015). Internet of things' connected devices to almost triple to over 38 billion units by 2020, dirección: http://www.juniperresearch.com/press/press-releases/iot-connected-devices-to-triple-to-38-bn-by-2020 (visitado 01-10-2016).

[2] Á. Villalba, J. L. Pérez, D. Carrera, C. Pedrinaci y L. Panziera, «Servioticy and iserve: A scalable platform for mining the iot», *Procedia Computer Science*, vol. 52, págs. 1022-1027, 2015, ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2015.05.097. dirección: http://www.sciencedirect.com/science/article/pii/S1877050915008972 (visitado 01-10-2016).

[3] PwC. (2015). Consumerization of apis: Scaling integrations, dirección: http://www.pwc.com/us/en/technology-forecast/2012/issue2/features/feature-consumerization-apis.html (visitado 01-10-2016).

[4] A. Dave. (2017). The 2017 docker usage report, dirección: https://sysdig.com/blog/sysdig-docker-usage-report-2017/ (visitado 10-04-2017).

[5] K. Wähner. (2016). The container landscape: Docker alternatives, orchestration, and implications for microservices, dirección: https://www.infoq.com/articles/container-landscape-2016 (visitado 10-04-2017).

[6] G. Inc. (2017). Google's v8, dirección: https://developers.google.com/v8/ (visitado 02-03-2017).

[7] A. Delgado. (2016). Top 5 companies using nodejs in production, dirección: https://www.linkedin.com/pulse/top-5-companies-using-nodejs-production-anthony-delgado (visitado 09-04-2017).

[8] U. Express.js. (2014). Understanding express.js, dirección: https://evanhahn.com/understanding-express/ (visitado 02-03-2017).

[9] D. Cartwright. (2005). Mysql 5.0 open source database, dirección: http://www.techworld.com/review/software/mysql-50-open-source-database-346/ (visitado 10-04-2017).

[10] R. Ho. (2012). Everything you need to know about couchbase architecture, dirección: https://dzone.com/articles/couchbase-architecture-deep (visitado 10-04-2017).

[11] T. Engineering. (2011). A storm is coming: More details and plans for release, dirección: https://blog.twitter.com/2011/a-storm-is-coming-more-details-and-plans-for-release (visitado 10-04-2017).

[12] A. S. Foundation. (2017). Apache storm documentation, dirección: https://storm.apache.org/releases/1.0.3/index.html (visitado 10-04-2017).

[13] I. Chenxi Wang. (2016). Containers 101: Linux containers and docker explained, dirección: http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html (visitado 10-04-2017).

[14] D. Inc. (2017). What is a container, dirección: https://www.docker.com/what-container (visitado 10-04-2017).

[15] C. Valdez. (2016). Why should i use node.js: The non-blocking event i/o framework?, dirección: https://developers.redhat.com/blog/2016/08/16/why-should-i-use-node-js-the-non-blocking-event-io-framework/ (visitado 02-03-2017).

[16] E. Hahn. (2017). Using http methods for restful services, dirección: http://www.restapitutorial.com/lessons/httpmethods.html (visitado 02-03-2017).

[17] M. Jones, J. Bradley y N. Sakimura, «Json web token (jwt)», RFC Editor, RFC 7519, mayo de 2015, http://www.rfc-editor.org/rfc/rfc7519.txt. dirección: http://www.rfc-editor.org/rfc/rfc7519.txt.

[18] E. Rescorla, «Http over tls», RFC Editor, RFC 2818, mayo de 2000, http://www.rfc-editor.org/rfc/rfc2818.txt. dirección: http://www.rfc-editor.org/rfc/rfc2818.txt.

[19] Couchbase. (2017). Database querying with n1ql, dirección: https://www.couchbase.com/n1ql (visitado 10-04-2017).

[20] D. Inc. (2017). Controlling startup order in compose, dirección: https://docs.docker.com/compose/startup-order/ (visitado 06-04-2017).

[21] R. Inc. (2017). Rancher compose, dirección: https://docs.rancher.com/rancher/v1.6/en/cattle/rancher-compose/ (visitado 08-04-2017).